Rana

Jørgensen, Søren Vissing

Rana

a flexible real-time agent-based simulation platform.

Søren Vissing Jørgensen

SDU

Center for BioRobotics - Mærsk McKinney Møller Inst.

November 29, 2016

Abstract

The aim of this work is the development of an open-source software tool, Rana, which enables real-time constrained simulation of multi-agent systems. Multi-agent systems research is a branch in the field of AI where focus is not on single agents, but rather on the emergent properties that arises from societies of agents.

Rana represents the event driven simulation. This means that it has a focus on representation of perceivable agent actions, called events. The Rana event is a flexible information construct that can be set to propagate across the environment in simulated physical time.

Rana's modelling paradigm offers flexible design of agent behaviour, which allows for separation of behavioural definitions for event handling and internal agent actions, both of which can be constrained by a real-time precision level. Each agent has a number af Rana specific modules at its disposal, these enable, collision detection, environment interaction, event generation and more. A number of demonstration agents is developed to illustrate the different facets of Rana agent design.

As a further expansion to the modelling paradigm an event processing function is introduced. Its purpose is to provide a common interface for defining the nature of an events propagation. This provides an interface for determination of event relevance during a simulation, and for visualization of a simulations event-scape. Two different simulations are presented that demonstrate event-processing functionality in agent design and for visualization.

As a tool Rana offers a graphic user interface for live event visualization, agent feedback and simulation configuration.

To establish Rana as an end-user tool, it is evaluated against the existing state of the art for multi-agent systems simulation tools and three different Rana models are presented: traffic, mining robotics and acoustic driven male chorusing.

Abstract, Danish

Målet med arbejdet presenteret i denne afhandling, er at beskrive udviklingen af et stykke open-source programmel ved navn Rana. Rana er et vætøj til design og simulering af real-tids multi-agent systemer. Multi-agent systemer er en gren indenfor kunstig intelligens hvor fokus er flyttet fra den enkelte agent, til de emergente mønstre der kan opstå i et samfund af interagerende agenter.

Rana representere den eventdrevne simulering, og har derfor fokus på representation af agenter og de events de udsender. I Rana terminologi er et event et udtryk, for et observerbart agent udtryk. Eventet er implementeret som en flexibel datakonstruktion der propagere i simuleret fysisk tid.

Ranas agent modellerings paradigme tilbyder fleksibelt design af agent adfærd, bland andet seperat definition af adfærd for både event opfattelse og de interne beslutnings processor. Den enkelte agent har i Rana adgang til et modulbibliotek, der blandt andet tilbyder kollisionsdetektering, miljømanipulation og generering af events. Der er desuden udviklet et antal agenter til demonstration af de forskellige agent design facetter.

Til yderligere udvidelse af Ranas agent design paradigme, er der introduceret en event-processerings funktion. Formålet er at tilbyde et interface til at beskrive funktioner der definer hvorledes et event propagere. Det muliggør adfærdsbaseret definition, af event relevans i simuleringssammenhæng samt visualisering af events og deres propageringsmønstre. To forskellelige modeller er udviklet til demonstration af event processoren.

Som brugsværktøj tilbyder Rana et grafisk brugerinterface til både live- og eventvisualisering, samt simulationskonfiguration og agent output.

Rana er desuden blevet evalueret mod den nuværende state-of-the-art indenfor generel multi-agent simulering, samt brugt til tre forskellige videnskabelige modeller: traffik simulering, autonome mine robotter samt akoustisk dreven kaldeadfærd i dyr.

Acknowledgements

The first, the one and only John Hallam, for him trusting me in my work even when I really did not know what I was doing, for his friendly demeanour, enlightening guidance and for fixing up my shoddy writing. I am eternally grateful for his support when I needed it the most.

Jacob Christensen Dalsgaard, for dragging me into the field and sending me to the US to wrestle alligators and for keeping my research grounded in reality.

Then there is my French connection, Yves Demazeau who has been a reliable source of invaluable input, support and criticism.

The best Linux guru and king of procrastination, my office mate of more than three years Thor Andreassen. It has been an inspiration to work alongside him.

The office chatterbox Leon Larsen. So many good discussions on basically every subject known to man, it is a wonder how we ever got any work done.

Mark Bee, for his hospitality and making me feel like a part of his lab from day one, and for the awesome trip to Texas, will be forever grateful for the help during my kidney-stone crisis. I have tremendous respect for his scientific view, and tenacious character.

Norman Lee, for his friendship and all the trips to Target shopping with food coupons, and for the discussions on family, career and mechanical keyboards.

Mikael Andersen, for a life-long friendship, and for helping me to keep it (sort of) real throughout this crazy ordeal.

Vilma, Bertil, Nova and Halfdan, my four brilliant children, best of the best.

Most importantly there is Nina, for still being my wife and believing in me even in my darkest hour.

My parents and in-laws, for helping with the kids and for their unwavering support.

Then probably the wisest man I have ever met Michael Greenfield, for helping me stay on course by supplying crucial knowledge at just the right time.

And then of course Bridget Hallam, the best constructive critic known to man.

Vibeke Nielsen, for her kindness and the nice chats and for ordering the pens I wanted. For helping out when rules became too rigid.

Birgit Davidsen for her patience and help all those time when I failed at administration tasks.

Finally, there is Arthur C Clarke, for providing inspiration, a guy who thankfully have produced a number of brilliant science fiction worlds that I never tire of visiting, indispensable when I need escape from the real science.

For Nina

Table of Contents

Abstract **iii**

Abstract, Danish **v**

Acknowledgements **vii**

1 Introduction **11**

1.1 Conceptualizing MAS 13

1.2 Emergence 14

1.3 Benchmarking 15

1.4 Real-time Considerations 16

1.5 Requirements 16

1.5.1 Tool Requirements 17

1.6 Achievements 18

I Rana 19

2 Introduction **21**

3 Design **23**

3.1 The Event 24

3.1.1 Definition 25

Table of Contents

3.1.2 Non-functional Definition 26
3.1.3 Properties 26
3.1.4 Conclusion 27
3.2 The Agent 27
3.2.1 Definition 28
3.2.2 Requirements 29
3.2.3 Properties 30
3.2.4 The Runtime Agent 31
3.2.5 Conclusion 31
3.3 The Environment 32
3.3.1 Definition 32
3.3.2 Representation 33
3.3.3 Discussion 34
3.4 The API 34
3.4.1 Definition 35
3.4.2 Extending the API With Runtime Modules 36
3.4.3 Conclusion 39
3.5 Timing 39
3.5.1 Event Handling Resolution 39
3.5.2 Internal Action Resolution 40
3.5.3 Conceptualizing the Steps of the Real-Time Simulation 40
3.5.4 Phases 41
3.5.5 Handling of Events 42
3.6 Design Structure 43
3.6.1 Structure Definitions 44
3.7 Non-Function Design Considerations 46
3.7.1 Multi-Core Support 46
3.7.2 Implementation Technologies 47
3.8 Discussion 48
3.9 Conclusion 48

4 Implementation **51**

4.1 The Event . . . 52

4.1.1 Event Representation . . . 52

4.1.2 The Event Reference . . . 53

4.1.3 Event Handling . . . 54

4.1.4 Valid Event Table Types . . . 58

4.1.5 Conclusion . . . 58

4.2 The Agent . . . 59

4.2.1 The Agent State . . . 59

4.2.2 The Agent Module . . . 61

4.2.3 The Lua Agent . . . 61

4.2.4 Movement . . . 63

4.2.5 Error Handling . . . 63

4.2.6 Conclusion . . . 64

4.3 The API . . . 64

4.3.1 Lua Module Support . . . 65

4.3.2 Conclusion . . . 65

4.4 Structure . . . 66

4.4.1 Interface . . . 66

4.4.2 Simulation Core . . . 69

4.4.3 Agent Domain . . . 70

4.4.4 API . . . 71

4.5 Threading . . . 72

4.5.1 Making API Access Thread Safe . . . 74

4.5.2 Concurrency and Behaviours . . . 74

4.6 Conclusion . . . 75

5 Demonstration **77**

5.1 The User Interface . . . 78

5.1.1 Control . . . 79

5.1.2 Live View 81
5.1.3 Conclusion 82
5.2 Agents 82
5.2.1 The Ping Pong Agent 82
5.2.2 Event Handling 84
5.2.3 Collision Detection 85
5.2.4 Data Collection 87
5.3 The Foraging Frog Agent 91
5.3.1 Agent States Via Modules 91
5.3.2 Experimentation 94
5.3.3 Conclusion 97
5.4 Discussion 97
5.5 Conclusion 98

II Event processing 99

6 Introduction **101**

7 Design **103**
7.1 The Event Processor 104
7.1.1 Visualizing Intensity 106
7.1.2 Conclusion 106
7.2 The Event-map 106
7.2.1 Intensity Processing 108
7.2.2 Saving Events 109
7.2.3 Intensity Representation 110
7.2.4 Conclusion 110
7.3 Design structure 110
7.4 Conclusion 113

8 Implementation **115**

8.1 The Event Processor 115
8.1.1 The Data Files 118
8.1.2 Conclusion 120
8.2 The Event Map 121
8.2.1 Z-block 121
8.2.2 Event-map Generation 121
8.2.3 Conclusion 123
8.3 Discussion 123
8.4 Conclusion 124

9 Demonstration **127**
9.1 The User Interface 128
9.1.1 The Event Processing Panel 128
9.1.2 Event-map Panel 130
9.1.3 Discussion 131
9.1.4 Conclusion 132
9.2 Event Processing 132
9.2.1 Agent Design 132
9.2.2 Results 134
9.2.3 Conclusion 134
9.3 Event Visualization 135
9.3.1 Agent Design 135
9.3.2 Event Visualization 137
9.3.3 Conclusion 139
9.4 Discussion 139
9.5 Conclusion 139

III Evaluation 141

10 Introduction **143**

Table of Contents

11 State of the Art Analysis **145**

11.1 Tool Evaluation 146

11.1.1 Repast 146

11.1.2 MASON 148

11.1.3 MadKit 149

11.2 Overall Evaluation 150

11.3 Discussion 152

11.4 Conclusion 153

12 Simulation of City Traffic **155**

12.1 Simulation Construction 156

12.1.1 Map Generation 156

12.1.2 The Agents 157

12.2 Results 158

12.3 Discussion 159

12.4 Conclusion 160

13 Mining Robot Simulations **161**

13.1 The Simulation 161

13.2 Results 163

13.3 Discussion 165

13.4 Conclusion 165

14 The Greenfield Model **167**

14.1 The Model 168

14.2 Translating the Model 169

14.2.1 Method 170

14.2.2 The Free-running Oscillator 171

14.2.3 Inhibiting Oscillator 172

14.2.4 Introducing the Phase Response Curve 174

14.3 Validation Experiments 176

14.3.1 Synchronization Experiment 177
14.3.2 Introducing the Free-running Oscillator (**I**) 179
14.3.3 Conclusion 181
14.4 Discussion 181
14.5 Conclusion 182

IV Conclusion 183

15 Requirements Evaluation **185**
15.1 The Modelling Paradigm 185
15.2 Simulation Interface 186
15.3 Tool Requirements 187

16 Discussion **189**
16.1 Rana 189
16.1.1 Parallel Discrete Event Switching 189
16.1.2 Event Propagation and Movement 191
16.2 Event Processing and Visualization 192
16.2.1 The Event Processor 192
16.2.2 Event Visualization 193
16.3 Evaluation 193
16.3.1 State of the Art 193

17 Future Work **195**
17.1 3D and Physics Support 195
17.1.1 Physics Integration 196
17.2 Environment Expansion 196
17.2.1 Weather 196
17.2.2 Map Support 197
17.2.3 The 3D Map 197
17.3 Simulation Configurability 197

17.4 Bridging Multi-channel Recordings and Modelling 198
17.5 Graphic Agent Design 199

18 Achievements **201**

19 Summary **203**
19.1 Rana Availability 204

Bibliography **205**

V Appendix 215

A Demonstration Appendix **217**
A.1 Modules 217
A.1.1 Optimizing API calls via a module 217
A.1.2 API Error checking via modules 218

B The Greenfield Model **219**

C Rana Demonstration Models **223**

D Testing **225**
D.1 Event Propagation Precision 225
D.1.1 Setup 225
D.1.2 Results 226
D.2 LuaJIT and Multi-thread Performance 226
D.2.1 Ping pong Test 228
D.2.2 Foraging agent results 228
D.2.3 Further Performance Testing with LuaJIT 229
D.2.4 Results 229
D.3 Conclusion 230

E Available Lua Module and C++ functions **231**

E.1 Module Overview 231
E.1.1 Agent 231
E.1.2 Collision detection 232
E.1.3 Core 234
E.1.4 Event 235
E.1.5 Map 235
E.1.6 Shared 236
E.1.7 Statistics 237
E.2 API functions 238
E.2.1 Output 238
E.2.2 Map 238
E.2.3 Shared Values 239
E.2.4 Physics 239
E.2.5 Simulation Variables 240
E.2.6 Collision Detection 241
E.2.7 Simulation Manipulation 242
E.2.8 Agent Manipulation 242

List of Figures **243**

VI Publications 247

Chapter 1

Introduction

Simulation of animal chorus behaviour, or other behaviour dictated by real-time constraints, be it internal neural delays or speed of sound delays, presents an interesting problem proposition in computer science. The goal for this dissertation is to offer a solution that provides a scalable approach to model, simulate and test proposed simulation targets.

The concept of real-time is a constraint that, in Nature, can be taken for granted. In the biological sciences everything is neatly synchronized by the central constant of time. However, in computer science, real-time or rather simulation thereof presents a problem for a number of reasons, some of these are:

- As opposed to Nature, duration of a given tasks in computer science is a dynamic parameter that depends on the following elements.
  - Operating system scheduling tactics.
  - Current system load, i.e. the number of tasks currently active on a system.
  - The number of CPU cycles it takes to perform the task on a given architecture.

  - The number of hardware threads and cores that are available.

- Heterogeneity between the time taken up by the simulated and biological task. For example, the time it takes for a simulated individual to take a new position can be near instant whereas it could take days in a real environment.

- Perception is always done in real-time, e.g. an individual cannot act on a sound or smell before it has reached that individual's senses.

- Simulation of real-time in a computer is represented as snapshots, whereas in we experience the world in continuous time.

Bridging computer-science and biology is a common theme in modern science and often application specific solutions are realized via generalized stochastic computational models [84] or distributed multi-agent systems (MAS) [40].

MAS can be used to imitate Nature at almost any level as it dictates a system that comprises a number of individual interacting agents. A biological MAS simulation can represent various levels of abstraction ranging from cells through individual animals to whole ecosystems. Currently there only exists a limited number of general tools that enable the design, implementation and simulation of such agents. In computer science these tools can be categorized as MAS simulation tools.

As previously stated Nature is constrained by a continuous real-time clock that is not directly translatable to computing systems on which the MAS simulation is run. So in order for MAS to be relevant for biological systems simulation some way must be found to make the simulated agents adhere to the constraints of a central clock shared by all agents.

1.1 Conceptualizing MAS

In order to offer a generalized approach for performing real-time constrained simulation of MAS, we first have to conceptualize MAS.

MAS is a concept in computer science that attempts to rid software solutions of the rigid paradigms of object oriented design [19]. MAS software design attempts to isolate potential self-sufficient entities that can be implemented and deployed as active subsystems. These subsystems are called agents.

Agents in a MAS can be simple reactive entities that act as black-box systems providing an output to some input. An example might be a sensor in a smart-home [10] which is usually a purely reactive entity. They can also be intelligent entities that interact and function within the real world; self-driving cars are a good example of this [75].

Multi-agent systems are distributed by definition: each agent is a separate entity. One of the more prominent examples of recent concepts that can be defined under the MAS paradigm is the Internet of Things [5], which is a concept where a multitude of small self-sufficient agents in a home or public place are connected to the Internet via the IPv6 protocol [14]. The devices are usually interfaced via a central application that controls and interprets information gathered by a number of separate agents. The patterns that can be exhibited by a multitude of such devices is in MAS called emergence [80]. Emergence in this example consists of data patterns of a number of Internet of Things devices, patterns that can be used to further understanding of a system and provide useful information for people and devices to which those agents have relevance. On a more abstract level, the term emergence can also be attributed to the patterns emerging from the organization of these devices [61].

1.2 Emergence

Decomposing a given system to a relevant level of agents and then deriving the behavioural patterns of those agents is an important task that serves to further the understanding of the system... especially considering technology's tendency towards support for multi-threaded hardware in single devices and a trend towards distributed systems such as as cloud services [3] and the Internet of Things. The same is true for biological systems that can be can be decomposed and abstracted to varying levels of agents.

When working with MAS we run into the concept of emergence. Emergence is the result of agents interacting with each other and their surrounding system. The problem with emergence is that it is very hard to predict the patterns that emerge when you thrust a number of self-contained agents into an environment.

In Nature the concept of emergence can be seen as patterns that emerges when a multitude of loosely or tightly coupled entities share an environment. For example, the emergent patterns involved in natural selection have arguably given rise to the biological systems we see today [13]. Emergence is usually observable if we know which pattern to look for. An example of this can be the flocking and flee mechanics of a small fish, such as a herring shoal [7]; another example is the emergence of an economic crisis, such as the financial crisis in 2008 [43].

Even if we know the pattern, understanding the cause and effect mechanics of the emergent patterns can prove to be a challenge. It gets even worse if we do not know which patterns to observe or if patterns emerge that have no immediate explanation.

When deploying a solution based MAS, such as the Internet of Things, to solve a task it is in the interest of the developer that, like the standard object-oriented approach, the system solves a the task via the emergent properties of the system.

A trait for solution based MAS software systems is that they are very

scalable. In a solution based MAS points of congestion can be removed by adding more agents of the congested type. So as long as an implemented system exhibits the emergent patterns decided on, everything is fine. However, like in Nature, emergence in software systems can be very hard to predict for the developer and end user, and adding more agents to a system might lead to unexpected emergence of undesired patterns [63].

That said, it can be easy enough to see the emergent patterns generated by adding an increasing number of passive agents; it is, however, not easy to predict the patterns that can emerge as a consequence of adding increasing numbers of intelligent autonomous agents to a system.

1.3 Benchmarking

Due to the potentially unpredictable nature of a MAS, it is in the developer's interest to benchmark the system to further understand the emergent properties it can exhibit. Benchmarking multi-agent systems is very much tied to the emergent properties of that system, giving rise to some challenges.

As it is difficult to predict emergent patterns, it might be prudent to adopt and develop general benchmarking paradigms. So the systems performance can be analysed prior to deployment, whether the system comprises smart home-/office agents that control lighting and heating or whether it is autonomous robotics, such as self-driving cars. However in science there currently exists no general way of benchmarking MAS.

For this dissertation we are mostly interested in the simulated real-time constrained MAS. Enabling benchmarking of a simulated system can be done in the following ways.

- **Data collection**: Individual agents can collect data relevant to their implementation. The agents can use this information to perform evaluation of their performance during runtime. This information can also be written to a data file for post-simulation analysis.

- **Observation**. Utility agents whose only purpose is to observe agents and emergent behaviour can be supported, for example data collectors that intercept agent communication and perform analysis of it. More general observation can also be offered via simulation visualization.

- **Visualization**. Graphics visualization of agent activity can provide a means for a simulation designer to gauge the emergent patterns, which can be seen by observing agent and environment interaction.

1.4 Real-time Considerations

With this dissertation the main focus is on simulation of biological systems. So we have to consider handling of problems where processing delays, both internal and external can play an essential role. Internally there can be neurological delays. Externally it can be sound propagation delays. Realistic simulation of physical time in MAS represents a challenge as it requires synchronization between the active agents.

The real-time aspect might contribute directly to a behaviour that dictates a prevalent emergent behaviour in a congregation of agents. For example, real-time is a very important aspect in regards to chorus mechanics where male animals chorus. A male whose call has precedence over fellow callers in the chorus is more likely attract a female via the the precedence effect [82]. The sorting through acoustic information via the precedence effect is in more broad terms called the cocktail party effect [4], which describes the ability for an individual to sort thought ambient chatter in a party and listen to single conversations.

1.5 Requirements

To enable MAS simulation of biological systems constrained by physical time we need to establish the tasks it entails. To do this we have established two

major initial artefacts, they are:

- **Modelling Paradigm**: The MAS simulation tool must offer a number of interfaces for designing agents and their interactions, both with an environment and each other.
- **Simulation interface**: The simulation interface should enable relevant configuration options and user interaction and observation of ongoing simulations.

1.5.1 Tool Requirements

With the general requirements in place we can establish a number of tool specific requirements for development of the tool, requirements that take into account all of the considerations from the preceding sections.

1. **Simulation of physical time**. To enable a fully featured real-time aspect we need to consider constraints of the following agent behaviours.
   - Agent actions, such as movement and sound emissions.
   - Interaction.
   - Information processing, or neural processing for biological agents.
2. **Agent modelling** that offers a suitable paradigm for agent design. For a tool such as this to achieve success, agent modelling should carry a relatively low learning threshold, especially considering the subject matter of real-time constrained simulations where research often is done by non-programmers.
3. **Benchmarking** of a simulations emergent properties. This can be done by allowing the user to inspect a simulation via visualization and user configurable data collection that generates data to be used for post processing.

The purpose of developing a new MAS simulation tool is to incorporate the real-time aspect, which is missing from the state of the art. This means that the tool has to be capable of controlling and tracking agents as well as their interaction in accordance with a central clock.

In Nature every agent adheres to the laws of physics and despite the fact that Nature is heavily distributed, all entities adhere to the constant of time. For an example, sound-based events... no matter the medium of propagation... propagate at a speed given by the laws of physics. To add to this natural agents are not able to perceive events that have yet to register at the location of that agent.

By fulfilling these requirements we can enable research into behaviours governed by time-constrained event propagation via scalable and flexible MAS simulations. Furthermore, regardless of the real-time aspect, the tool can possibly also be used for more general MAS simulations.

1.6 Achievements

During the work on this dissertation the following achievements have been targeted.

- Successful fulfilment of the requirements for development of a real-time MAS simulation tool.
- Offer a feature complete tool that is in a complete state with a reasonable level of documentation, enabling it to be put into the hands of third parties for use in various projects.
- Demonstrate interfacing of biological modelling and MAS using the developed tool.
- Provide models and demonstrations that provides evidence towards the tools diversity and usefulness.
- Scientific recognition of the tool in the field of computer science and biology.

I

Rana

"The prospect was dazzling. Many also found it terrifying, and hoped the enterprise would fail. But they knew in their hearts that once science had declared a thing possible, there was no escape from its eventual realization"

Arthur C. Clarke, Childhoods End

Chapter 2

Introduction

A curses [68] based real-time constrained multi-agent simulation tool was conceived as part of an M.Sc. project [36] in the spring of 2013. That tool represents a prototype that set the foundation to develop a new fully featured end-user accessible MAS simulation tool, for the purpose of simulating real-time constrained biological systems.

This new tool is named Rana after a genus of frog, as it was research on frog chorusing, specifically by Douglas L. Jones and Rama Ratnam [35], that inspired its conception and defined some of its core requirements.

Here the story of Rana's development is described in the following three chapters: design, implementation and demonstration.

Chapter 3

Design

In the dissertation introduction the subject matter of interfacing MAS simulation and biological systems was conceptualized, which lead to a set of requirements that have to be fulfilled in order to offer a real-time platform-independent MAS simulation. This leads us our next target which is addressing the various design criteria required for development.

First we will conceptualize the event driven simulation that we want Rana to represent. This is done by detailing four critical artefacts of the Rana simulation.

- **The Event**. Arguably the most important artefact in the real-time simulation. The event is a representation of an agent's external actions.
- **The Agent**. The active entity, a free agent with a pre-programmed behaviour.
- **The Environment**. The interactive representation of an environment.
- **The API**. Provides a wide range of functions that, among other things, enable the agents to interact with each other and the environment.

I. Rana

Once the four artefacts has been established the issue of timing will be dealt with. Then the design of the structure of the tool will be discussed along with a description of its various sections. Finally as a lead-in to the implementation we will discuss a series of non-functional design concerns that establish some technological challenges and solutions towards building the tool.

3.1 The Event

Consider an environment with two male frogs exhibiting antiphonal mating call behaviour. This type of frog is interested in alternating its calls with fellow males as a strategy to attract females [41]. Antiphonal behaviour is driven by each male optimally fitting their calls in between the other's calls.

In a simulation we need some way of representing the call. In other words, we need a way for agents to signal each other. This can be done by enabling agents to observe each other's behaviour; however, that is not computationally feasible in regards to scalability. Furthermore, observation mechanisms will become very complex in terms of timing as well as the logic required to determine which attributes to expose for observation. Rather than constant inter-agent observation, the tool should offer a way for agents to push interrupts or rather notifications to each other. So when a frog agent emits a call, the other frog agent receives a notification.

It is not enough just to have a notification system, as we also have to consider the real-time aspect. If the frogs in the previous example are spaced relatively far apart, sound propagation can affect the strategy needed for the individual frog to space its call in between other calls [25]. So we also need to give the notifications a real-time propagation property. But real-time propagated notifications are not enough in the case of frogs. Frogs will internally process sounds based on duration, frequency spectrum and intensity [50]. To enable this notifications have to contain the information required for the type of notification they represent.

Finally we also need to consider the nature of the notification. It is not enough that the agent submits information-heavy notifications that propagate in real-time. The frog's call is a sound and sound is broadcast by default. Peer-to-peer sound emission is not a realistic scenario. In nature female frogs or predators can and will listen for the calls of the frog [78]. So the tool needs to enable reception of emitted notifications for all agents in a simulation. The frog should, by design, not be able to control which agents that notice its call.

In Rana we have chosen to call these notifications *events*.

3.1.1 Definition

Events are emitted from the agent's point of origin with a predefined propagation speed e.g. a frogs call will have a propagation speed of approximately 343[m/s]. For visual displays, events should have near instant propagation.

Incoming events can go through a number of processing nodes on the receiving agent. For sound they can be neurologically filtered based on sound intensity, frequency spectrum and precedence [50]. The filtering is in place to remove irrelevant sounds. It is important that the receiving agent can make a decision on which sound it will react to.

Events are, once emitted, immutable by the emitter.

It is also prudent to interpret external actions other than calls as events. These could be visual cues such as threat display [62], ground tremors or even radio signals. Thus we define the event as a representation of any perceivable external action of the agent. This external action should, in most cases, be public so they can be perceived by all other agents active in a simulation.

While we can consider events as public there are cases where event based targeting can be beneficial. In a multi-species environment there might be agents that are not able to perceive a certain type of events. For example, In an environment with bats hunting frogs the frogs will not be able to hear the high frequency calls emitted by the bat, but other bats in the simulation will.

A simulation could then be optimized by ensuring that the bats' call events are only broadcast to other bats.

3.1.2 Non-functional Definition

Now that we have established the event as a representation of external actions we can consider some secondary parameters to make the event even more useful as a MAS simulation artefact.

Multi-agent systems are notoriously hard to benchmark. Events represent a good opportunity to further the understanding of a simulation. Events can therefore be extended with an extra attribute that allows an agent to transmit the event as a peer-to-peer notification for the purpose simulation support. In the case of the frogs, their agent representations can have a support function that collects data on call frequency and call timing. This data gathering function can then sent the data collected to a central data processing agent that can generate statistic analysis of the results from the frog agents.

3.1.3 Properties

With event functionality established we can define a set of default properties, they are.

- **Propagation Speed**: In metres per second; the tool can allow a speed of 0 which means near instant distribution.
- **Description**: A string descriptor, allowing receiving agents to categorize the event.
- **Table**: A data table, useful for data collection or more advanced agent information exchange.
- **Target ID**: Single target event, a value that when set will ensure that the event is sent only to the target agent.

- **Target Group**: For targeting a specific group. For example the bat can belong to a high frequency group.

Each of the event's attributes should hold a default value, so the agent only need to define the ones relevant for the simulation.

3.1.4 Conclusion

The event is a powerful concept both as an external action of an agent and as an internal data structure used sharing benchmark data. For our male frog example where the agent behaviour is primarily driven by external events, the event mechanics design represent a very important behaviour design artefact.

3.2 The Agent

Another integral artefact is the agent itself. In the case of our two male frogs each frog is an agent whose inner workings is what ultimately defines when they emit a call and what happens when they hear an external call.

Most species of frog only register an external call if they are not calling themselves, since they cannot hear when they are in the middle of a call. If they register an external call some neural processing is done that takes several milliseconds [28]. Furthermore, if the frog decides to act on the call by emitting its own, it needs to prepare its physiology for call emission, all of which takes time. In a simulation all of these delays in the various stages of the frog need to be addressed. This requires an agent design paradigm that can handle these delays as they can occur dynamically throughout a simulation.

But that is not enough. While response to external actions is important, some frogs will need to start calling sporadically at some point or the simulated frogs will never start a chorus. In Nature, the first call can be triggered by observation of the time of day coupled with position and some individual propensity for initiating a call [29]. In a simulation, call propensity can simply

be implemented via a propensity attribute that increases the likelihood for a call over time when no external calls are registered. To prevent individual frog agents from exhibiting too similar calling behaviour, stochastic variables can be introduced to create varying call profiles. What this requires is that agents in a simulation are queried for an action at reasonable intervals.

There is also the general behaviours of the agents to consider. If we want to introduce a bat hunting the calling frogs we need some way of implementing individual agent behaviour; stochastic variables can only do so much. So a simulation should support simultaneous separate agent implementations.

Aside from interaction and observation of actions, the agent should also be able to change position and move about in a simulation. This would enable the frog agent to move to a different perch and allow bat agents to fly around hunting frogs.

3.2.1 Definition

First, the agent's representation in the simulation is defined. In a MAS the agent is typically a separate program or object that interconnects with other agent objects on some level [52]. So the frog and the bat agents of our previous example can be represented as separate programs that interact via events.

To enable agent interaction via events the agent needs to be able to react to incoming events, via a reaction or rather an event-handling function.

Upon initialization of the simulation the agent will be given the ability to define itself e.g. the frog would probably like to define its call propensity value through some stochastic variable as well as its starting location within the simulation environment.

Then there is the machinations of the agent as an individual such as the internal decision making process and movement. This can be done by allowing the agent to process and take actions via a function that is executed at reasonable intervals, which could every simulated millisecond.

It would also be prudent to have some way of enabling agents to detect each other as individuals. This can enable collision mechanics which means that an agent can detect and avoid other agents without resorting to events. It will also allow for simulation of vision so agents can scan for other agents and determine their type and position.

Finally, each agent should be given the option to wrap up their behaviour at the end of the simulation. This could entail writing out collected data for post processing.

3.2.2 Requirements

A set of requirements for the agent can now be established. The agent will be a self sufficient program that needs the following functionality.

- **Initialization**. Enables the agent by initializing behaviour-critical variables.
- **Handling of events**. Enables handling of events the moment they are perceived.
- **Internal action**. Allows agents to take some internally generated action such as moving and emitting events.
- **Finalization**. When the simulation is done, agents can be allowed a final set of actions, mainly for data processing purposes.
- **Movement**. It will be possible to set a destination for the agent and a speed which can enable the agent to move in real-time with the precision equal to internal action precision level.
- **Collision Detection** The agent is able to update a its position within a collision grid whenever relevant, this will allow the agent to be visually detected by other agents.

3.2.3 Properties

A number of properties are provided to support synchronization and information exchange between agents. It is possible to differentiate between two types, mutable and immutable. Mutable variables can be synchronized between the agent and the simulation core at every step. Immutable variables are simulation-specific constants such as environment size.

- **Mutable**
  - **Position**. The position of the agent.
  - **Destination**. Current destination of the agent.
  - **Speed**. Movement speed of the agent in meters per second.
  - **Collision detection**. Enable and disable collision detection.
  - **Moving**. Denotes whether the agent is moving.
- **Immutable**
  - **Id**. A unique runtime id of the agent that allows agents to identify each other and also allows them to identify the source of events.
  - **Environment size**. The size of the environment which is the width and height of the environment.
  - **Step resolution**. Denotes a precision level for the agents internal actions. It denotes how many times the agent will be queried for an action per second. It also denotes the precision level for movement.
  - **Event resolution**. Precision level of event distribution, the precision at which agents receive events.

The two resolution parameters are a representation of how Rana handles the timing of the agents. Timing will be discussed in section 3.5.

3.2.4 The Runtime Agent

Rana's implementation language is a compiled programming language, C++ [69]. By extension, agent behaviours will have to be compiled along with Rana. So Rana would have to be recompiled with every new behaviour an agent designer would like to test. This is not feasible for a number of reasons.

- Compilation is not task for regular users. It cannot be assumed that all people that have an interest in simulations are computer scientists. It also requires a whole set of secondary tools to compile a program and compilation for a complex program such as Rana can take quite a while.
- Compiled languages are often quite complex and rigidly defined, which is especially true for Rana's C++ code base. Setting an unreasonably high threshold for programming knowledge is not an attractive proposition. This would exclude whole groups of scientists, such as biologists who, depending on their field, can be experts on specific systems where MAS simulations are relevant.

To remove the need for compile time agent design we have chosen to interface Rana's simulation core with a dynamic programming language called Lua [31]. Lua is a fully fledged programming language that offers a very efficient way of interfacing with a C++ program. This moves simulation design to the agent itself: as agents can be implemented and tested dynamically during runtime, the user will never need to compile or re-compile the simulation tool.

It is also very important to note that events are defined at the agent level, so events are emitted by the runtime agent which allow events to be designed in runtime too.

3.2.5 Conclusion

The agent is the actor of the simulation. It is a self-contained entity capable of moving, handling and emitting events and taking internal action. By estab-

lishing the agent we now have a definition of the two most important artefacts in place: Events and Agents. The runtime agent interface; which separates agent and event design from the simulation tool, has also been introduced.

3.3 The Environment

With the event and agent artefact in place we can define the final artefact, the environment. Environmental factors can play an important role in regards to the behaviour of agents. Again we can consider the chorusing frog. In nature, most species of frog prefer to call in a watery environment due to females laying their eggs in the water, as tadpoles are wholly aquatic (there are exceptions though [45]). Not only that but some frogs, such as the Natterjack, prefer to sit on the shore while others, such as the green tree frog, prefer to sit on reeds and in bushes. To simulate this, some representation of the environment needs to be decided on.

Agents can be used to represent almost any active or passive actor in a simulation. This means that they can also be used to represent environmental elements such as plants and rocks. And while agents are an option for representing passive environmental actors they are not necessarily the best choice as they can present challenges both in design and computing complexity. Needing thousands of agents representing largely passive reeds in a green tree frog simulation could prove needlessly complex.

So rather than having agents represent every aspect of the environment a different approach could be to offered: a grid of position dependent attributes that can be used to describe sections of the environment such as water, sand and grass.

3.3.1 Definition

The environment is mostly a passive component that agents can use for navigation and interaction. It is passive in the sense that it will not have a predefined

behaviour, and its meaning is whatever the agent behaviour decides.

Agents should be able to manipulate the environmental attributes. This enables implementation of utility agents which generate a suitable environment on agent initialization e.g. for a simulation of Natterjack toads chorusing a lake environment with shores could be generated.

3.3.2 Representation

To fulfil the map definition the environment can be represented as a 2 dimensional image. To lower the implementation threshold we have chosen to utilize a lossless bitmap image format. This makes it possible to represent the environment as a 32 bit RGBA (red, green, blue, alpha) map entity which consists of four 8 bit channels each holding a value between 0 and 255. This bit-mapped environment can provide a simple and powerful environment representation for agents.

A use-case for the bit-mapped environment can be the following. A Natterjack chorusing simulation needs to be designed. First a map is generated via a set of random geometric shapes by a utility environment agent that generates a bitmap suitable for the simulation. The bitmap generated can have three different environment properties each with a different colour value, green for grass, brown for the shore and blue for water. The Natterjack agents can then move around and read the map, and once they have moved to the brown bitmap value they can settle and be ready for chorusing.

Outside ease of implementation the bitmap representation has an observational advantage, in that its values can match the visual representation that it seeks to emulate. So a lake can be blue, grass can be green and a shore can be brown.

Interaction with the environment requires an agent interface that allows for manipulation and surveying of environment variables both on initialization and during the simulation. This interface do not necessarily need to be event-based, but rather it could be some general function available to the agent, via

an API interface (such an interface is in place for Rana and its design will be described in section 3.4).

3.3.3 Discussion

The bitmap approach to the environment is rather straightforward but it does represent some limitations. A single bitmap is a 2 dimensional entity and as such it can be a convoluted exercise to represent attributes such as temperature and pressure on a single map. What could be implemented in some future iteration was to have several bitmaps one for each simulation relevant environment attribute such as the map the level of sunlight and the temperature.

Essentially using a single image for environment is a hack. The environment really is a function mapping place to properties and attributes and the image is a cheap way to do it.

The bitmap environment approach is limited in function but it does represent a good first step towards a useful environment type both for agent interaction and user presentation.

3.4 The API

With the three simulation-critical artefacts defined we can move on to the artefact that allows us to tie it all together, the API.

Even the simplest simulation with the two calling frogs requires some interface to allow for event emission and generation of the stochastic variable that would determine call propensity. In Rana we call this interface for the Agent Programming Interface, or API.

Agents are basically self-sufficient programs. So they will need a common interface that can provide them with a number of functions to enable them to represent the MAS they are meant to simulate. The API is that interface.

The API has to fulfil a wide range of design criteria, spanning from environment manipulation to event emission to interface output. Moreover the

API is not just an interface to serve the simulation it also has to solve a number of non-functional requirements such as allowing agents to output status messages to the user-interface that can give a user a clearer picture of what is going on in the simulation.

3.4.1 Definition

Due to the API having to support a wide range of functions we can split it into several separate independent sections. This has two advantages: it allows us to define clear definitions for each section, and from a computational point of view each section can be accessed in parallel which will carry a performance advantage when executing agent behaviours in separate hardware threads.

Representation of the following simulation specific sections for Rana has been chosen.

- **Map**. For reading and writing to the 2 dimensional bitmap environment.
- **Physics**. Allows for calculation of distance, retrieval of current time and other simulation data also allows for generation of random numbers using a central seed.
- **Scanning** Provides fast scanning algorithms to retrieve masks, which are matrices which denote valid values for scanning at a certain shape, this enables an efficient way to simulate fields of vision both for agent detection and environment scanning.
- **Shared**. Gives access to data repositories for sharing numbers, strings and data tables. This allows agents to share data with other agents. This data could be global variables, such as overall temperature or the Id of the agent that controls the environment.
- **Collision**. Provides access to a collision grid, which holds information on positions of all active agents that have collision based movement enabled.

The collision grid will have adjustable precision. A precision level of 1 means that each section of the grid is 1 by 1 meter, and a level of 0.01 is equal to a grid precision of 0.01 by 0.01 meter.

- **Agent**. Allows for the agents to remove and add agents e.g. the bat can hunt a frog and if it catches one it can remove it from the simulation. It also enables agents to change their colour representation as well as joining and leaving event groups.
- **Events**. Enables emission of events.

The API also has the following non-functional sections.

- **Output**. Allows the agent to output text messages to the user interface. Two functions are supported, one with suppressible debug messaging, another for regular output.
- **Core**. Allows for agents to stop a simulation.

A comprehensive section of available API functions is listed in the appendix, section E.2 on page 238.

3.4.2 Extending the API With Runtime Modules

Adopting Lua as our agent design language brings with it another advantage, which is runtime modules.

Via the Lua engine we can expose the API functions directly to the Lua agent. But Lua gives us the ability to provide the agent designer with a much more complete set of tools as the Lua module approach can be used to expand the API and simplify agent design.

Modules provide a powerful interface which allows us to take advantage of the runtime advantages Lua brings such as, dynamic function calls and type checks which offer more comprehensive error checking during agent testing and debugging. Modules can also help reduce the number of redundant calls

to the API by storing local API specific values. For example if the agent wants to request a colour change the module responsible can hold a local value and check whether the requested colour is different from the current thus potentially saving an API call to change the colour.

Modules can interface with the API in the same way the Lua agent can. To support this module approach Rana has a number of support modules that serve to extend the API. Figure 3.1 displays the current state of the interconnectivity between the provided modules and the API.

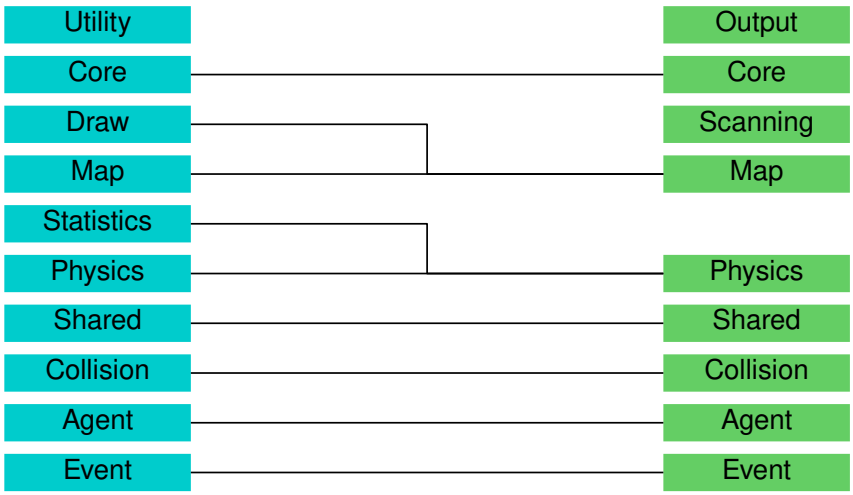


Figure 3.1: Module and API dependency diagram. **Lua Modules** **API**.

The following Lua modules are offered.

- **Utility**. Has the ability to serialize and de-serialize data tables, mostly used for debugging purposes. This enables the agent to write out event data tables for inspection.
- **Core**. Mainly for stopping the simulation and retrieving simulation-specific data such as the current simulation time.
- **Draw**. Provides a base set of tools for drawing shapes on the map.
- **Map**. Allows the agent to change and read individual pixel values for the map.

I. Rana

- **Statistics**. Provides an interface for retrieving various stochastic values.
- **Physics**. Calculation of distance and other physics based functionality.
- **Shared**. Allows the agent to store runtime data tables, strings and numbers. Will automatically serialize table data into strings upon storing and de-serialize upon retrieval.
- **Collision**. Provides the ability to check positions for the presence of other agents, update positions both asynchronously and atomically, as well as performing radial scans for other agents.
- **Agent**. Same functionality as the corresponding API section but with error checks.
- **Event**. Offers a dynamic interface for emitting events.

Modules also come with a couple of secondary advantages in relation to MAS agent design.

- **Design granularity**: Modules offer a way in which users can develop type-specific libraries that ease implementation and experimentation for specific agent types. For example for frogs a frog module could be developed.
- **Ease of functionality expansion**: as with type specific modules, it should also be possible to develop and offer functionality-specific modules that can range from environment variable generators to on-line stochastic analysis modules.

A comprehensive list of API functions are listed in the appendix, section E.1 on page E.1

3.4.3 Conclusion

The API supported by runtime modules is the last of our design artefacts. As previously stated it is what ties the previous three artefacts together. It presents a modular design to Agent functionality where each section is easily extended and new sections can be added.

3.5 Timing

With the final design artefact in place the timing of the real-time simulation can be addressed.

The goal for Rana is to enable true-to-real-time simulation of MAS. Simulation of physical time in computer science is not possible without adopting some level of discretization. For Rana, it is possible to split the simulation time flow up and offer two levels of precision: a very high resolution for the event handling and a lower resolution for internal actions.

3.5.1 Event Handling Resolution

Consider once again of the acoustically driven frog chorusing simulation. It is possible to have very high density choruses in nature where each individual needs to process calls from many sources at some level within a very short period of time. There are some natural attenuation modifiers such as sound intensity but mostly each agent must sort through incoming sounds internally. This requires a high level of granularity on event distribution, as the time of event reception is a very important property for this [82].

Luckily, due to the sporadic nature of events, it is possible to offer a very high level of precision for event distribution, as we only have to consider an events time of arrival at an agents position and all other times are irrelevant for event-handling purposes.

3.5.2 Internal Action Resolution

Sadly, it is not possible to just skip to the next time an event needs handling; we have to implement a second set of timings as the internal actions for agents also need to be taken into account.

As has been previously established, some form of agent action initialization is needed. This can be used to simulate either delayed reactions to events or actions based entirely on internal processes. Delayed event reactions are usually the more realistic theme in biological simulations. In the case of the chorusing frogs, there usually is a 10-15[ms] delay before the individual is able to physically formulate a response.

With this in mind we need a way of ensuring that agents will be queried with a reasonable level of precision.

It is also prudent to use the precision level of an agents internal actions to determine movement granularity. A good way of handling this would be to enable individual agents to define a precision level that determines at what resolution they will be queried for an action. Doing this would allow them to become purely reactive agents, which can be be useful for support agents such as data collectors.

For this to work the simulation tool needs a base precision level that determines the lowest possible action precision level.

3.5.3 Conceptualizing the Steps of the Real-Time Simulation

With the need for two different resolutions established, it is possible to embrace the following concepts that allows for synchronized simulation of real-time MAS.

- **Two-Tiered Agent Precision**: Agent behaviour operates with two different precision levels, one for internal operations, another for handling of external events.

- **Action based simulation flow**: In Rana things only happen when either an event propagates to a relevant agents position or when an agent is queried whether it wants to perform an action. This, along with the two-tiered precision level ensures that simulation run-time can remain reasonable even with heavy agent activity.

- **Wait for Agent**: To prevent timing issues the simulation core will wait for each individual agent to complete its current action. While this approach might seem counter-intuitive, it prevents the need for time reversals and de-sync handling [cite].

3.5.4 Phases

Simulation flow is split into phases that each represent a phase in the life of the agent. The agent can then implement a function to handle each one, which will make for a clean and clearly-defined agent design paradigm.

The simulation phases are the following.

- **Initialize agent**. Initializes the agent, allows the agent to perform initial actions such as setting up simulation specific function, initializing other agents, change to a desired start position etc.

- **Take-step**. Allows the agent to initiate new actions, move and perform delayed processing of event data.

- **Handle-event**. Event handling can, due to its more sporadic nature be distributed with great precision. So propagation and reception of events can be done with greater granularity than the take-step phase.

- **Clean-up**. When a simulation run has completed, or if its stopped prematurely either through the API or via user interaction, this is the function that will be executed. This allows for an agent to perform a final set of actions, such as processing collected data.

3.5.4.1 Take-step

This phase is the most significant for the agent as it provides opportunity for agents to take an internal action and synchronize with the core.

For biological agents, the the precision level of agent actions is usually at the level of a couple of milliseconds; robotic agents might have faster processing time, depending on the robot and the task.

This phase can, at very high levels of precision, potentially be very demanding since agents need to queried for an action at every step. So a take-step precision of 1[ms] means that each active agent is queried for actions 1000 times for every simulated second.

Agent movement and synchronization with the simulation core is also handled in this phase.

3.5.4.2 Handle-event

This phase is initiated on the individual agent when an event has propagated to its position. In acoustic driven simulations this phase is sporadic as agents will handle the events based on their position relative to the point of event origin. If the event propagation is instantaneous the event is handled right after the currently active phase on all recipients.

As the number of handle-event phases is typically an order of magnitude less than the number of take-step phases, the precision level for event handling can thus be much higher. This allows for agents to determine which event propagated to their position first. For example, this can be used for simulation where the cocktail party effect [59] can affect agent behaviour.

3.5.5 Handling of Events

Finally, some mechanism has been been put in place to handle the event handling timing. In Rana the mechanism responsible for this is the event handler. The event handler is basically an advanced container, that defines events and

enables event distribution. If we consider the two calling frogs from earlier we can outline the processing of one of the calls.

- Frog 1 emits a call event.
- The event is submitted to the event handler.
- The event handler calculates the time of arrival at frog 2's position.
- The event handler stores reference has an activation time that corresponds to the arrival time as well as a reference to frog 2.
- The event handler stores this activation time in a queue, so that Rana's simulation core will know when to activate the event.
- Once the simulation time reaches the stored time, the event handler will forward the event data to Frog 2 which can then run its handle-event function.

The event handler uses the above approach to keep the simulation core updated when the next relevant event handling time is up. This is the mechanism that allows skipping non-relevant times and thus achieve a very high level of event handling precision without excessive book-keeping.

3.6 Design Structure

Now that the artefacts of the Rana simulation have been established, the structure of Rana can be built. The program structure has largely been derived from the classic domain model [32], with some departures. The structure is designed to take all design concepts into account with as few compromises as possible. The structure is displayed in figure 3.2.

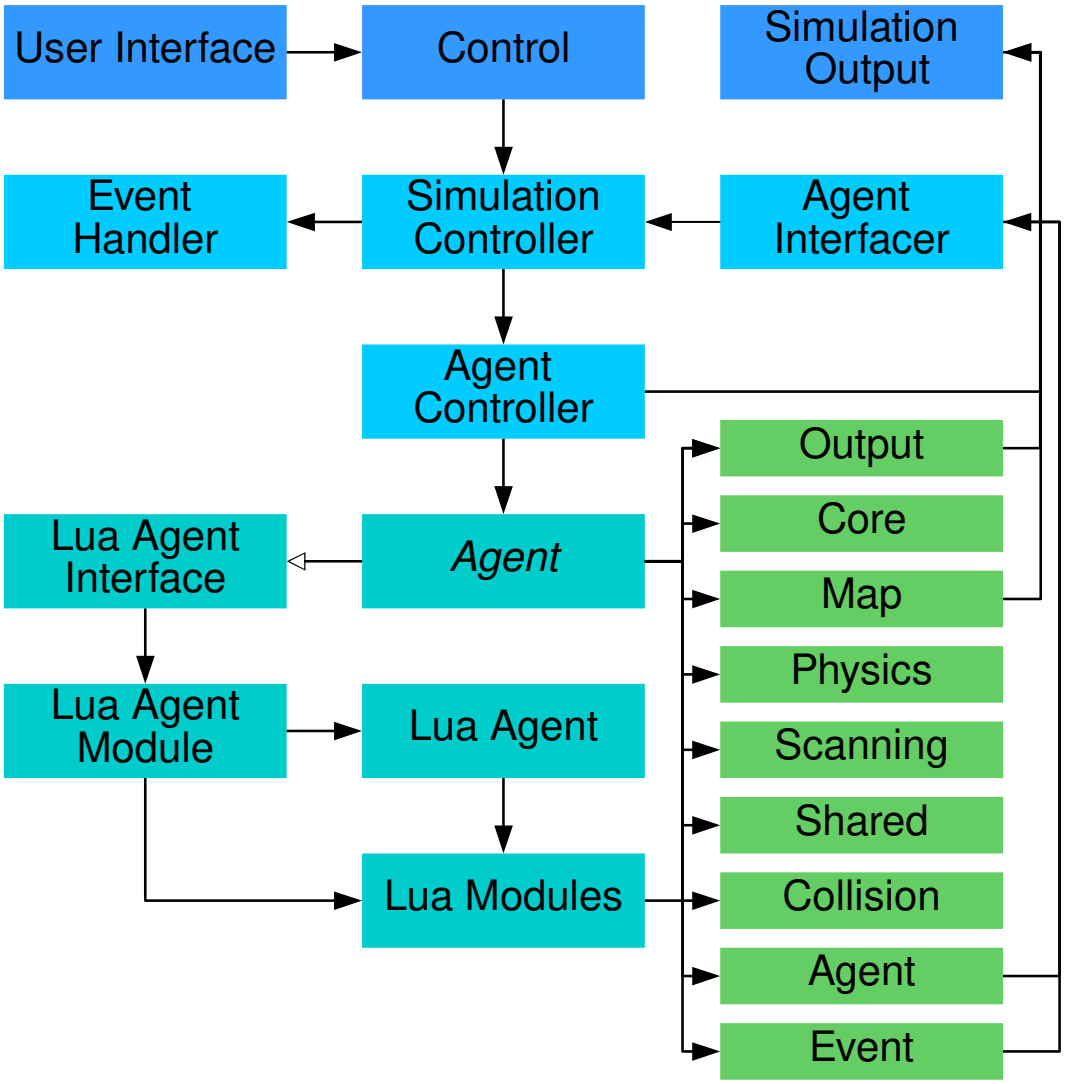


Figure 3.2: Rana design structure diagram. ■ **Interface**, handles user input and simulation output. ■ **Core**, handles simulation flow. ■ **Agent Domain**, agent interface for agents their initialization and API interaction. ■ **API**, asynchronous application programming interface for agents.

3.6.1 Structure Definitions

■ **Interface**. The interface of Rana handles all user input and handles the display of all agent and graphic output, it consists of the following parts.

– **User Interface (UI)** is a classic mouse driven input that provides a suitable amount of control for the user, to enable running simulations with various parameters.

  - **Control** Will instantiate the core and pass user information to the core, both during simulation and upon simulation initialization.
  - **Simulation Output** Provides live simulation visualization as well as the text output from agents.

- **Core**. Handles simulation flow and agent interaction via events.
  - **Simulation controller**. Has simulation logic, determines the time of the simulation, ensures synchronization between the agents.
  - **Agent Controller**. Is responsible for a number of agents, signals agents when there is an impending event or the next step phase is up. The simulation controller will instantiate a number of agent controllers equal to the number of desired runtime threads.
  - **Core Interfacer**. Allows the API to interact with the simulation controller to retrieve the current time, stop a running simulation or submit events.
  - **Event Handler**. Handles the events, denotes the next time an event must be activated by an agent.

- **Agent Domain**. Contains the various parts needed for agent implementation and agent API interaction. All classes here except for the Agent and Lua Agent Interface are run-time implementable.
  - ***Agent***. Abstract class that defines the agent variables and functions. It also provides interfaces for the various simulation phases.
  - **Lua Agent Interface**. Pre-compiled Lua agent representation that is derived from the agent class. Sets up the Lua agent state and synchronises the global variables that denote movement, position etc. It also provides static methods that allow the Lua Agent to interact with the API.

I. Rana

- **Lua Agent Module**. Provides a secure interface for the Lua Agent. Provides default templates for all simulation phase functions. Allows for dynamic Lua Agent modelling and provides error checks on implementation.
- **Lua Agent**. The runtime agent representation.
- **Lua Modules**. The runtime modules, that provides dynamic access to the API.

- **API**. The API provides a wide variety of functions agents such as event emission and environment interaction.

3.7 Non-Function Design Considerations

Before we delve into implementation a number of secondary design concerns will be be addressed: multi-core support in relation to the real-time MAS simulation, and the choice of implementation technologies.

3.7.1 Multi-Core Support

To enable real-time simulations in MAS it is not possible to give each agent a free running thread without compromising results. It is a requirement that simulation outputs are consistent across platforms. Results should not be a representation of the system on which the simulation runs but rather the implemented behaviour of the agents.

Since modern computing environments have two or more processing threads it is therefore a priority in the design phase that we incorporate support for multiple simulation threads, so the simulations can take advantage of multiple threads.

3.7.2 Implementation Technologies

The current state of the art is leaning heavily towards Java as a base programming and agent design language. Java has the advantage of offering cross-platform compatibility out of the box. Cross-platform capability is a clear advantage if wide distribution is desired.

While we do not target Java as the main back-end technology we still intend the tool to support a variety of platforms such as Linux, Windows and MacOS. An alternative for high performance cross-platform implementation languages exists in the newer iterations of C++.

These are the languages that will be utilized for implementation.

- **C++**: The newer standardizations have good cross-platform support. C++11, has via the Standard Template Library [49], platform independent interfaces for multi-threading.

- **QT** [73]: A C++ based framework that offers a suite of cross-platform development interfaces; has seen heavy adoption by the open source community mainly due to its powerful GUI toolbox.

- **Lua**: A scripting language, originally developed to ease configuration of devices in C++ programs. Lua has proven itself to be a fast and versatile scripting language, its speed is furthered by the externally developed LuaJIT [56] interpretation engine. As a dynamic programming language Lua offers run-time dependent agent design language, which means that there will be no need for re-compilation of Rana.

- **LuaJIT**: Externally developed interpreter that offers Just-in-time compilation of Lua scripts. Adopting LuaJIT can greatly speed up agent behaviours.

3.8 Discussion

Agent actions are discretized which means that agents, aside from event reception, experience the world through a series of snapshots. This can present some challenges if two agents decide to manipulate the exact same section of the map at the exact same time-step. Depending on the behavioural design of the agent this could possibly lead to a never ending flickering back and forth between the two agents [47]. Such a scenario can, in one case, be prevented by introducing an environment control agent, so when agents want to change a section of the environment they transmit a request in the form of an event and the environment agent can then sort the conflict out.

There is an inherent problem with the way event propagation is handled in Rana. This is due to event arrival times being calculated at the time of event emission. This tactic enables near infinite precision with no real performance impact. The problem comes if an agent moves very fast in relation to an agent that emits a sound based event. The real time of arrival will not match the one the simulation engine has calculated. This is something that needs to be considered when designing a simulation. The problem can be alleviated by submitting time critical events with instantaneous propagation time that will allow agents to track the events activation time depending on their position using the simulation take-step precision.

3.9 Conclusion

Rana's design, in particular the structure and API, has gone through many iterations throughout development. The current iteration presents the solution we think offers the most optimal approach to real-time MAS simulations.

The design process has throughout development revolved around satisfying the need for enabling the acoustic driven biological simulation. We have been especially focused on chorusing modelling. The constraints for enabling such

a simulation have proven to provide us with a very versatile design that can satisfy many other subject fields, as will be demonstrated in the following demonstration chapter.

Chapter 4

Implementation

Implementation of Rana has been done primarily to enable the tightly timed event driven simulation concept presented in design. This chapter will address all significant implementation details of the four artefacts. The runtime agent design aspect will be the main focus as it is our primary target for agent design.

During implementation we maintain the four artefacts established in design.

- **The Event**. Again we will start with the event. This time with a description of the event representations and the handling of the events.
- **The Agent**. Details the task of enabling of behaviours and the mechanics of the runtime agent.
- **The Environment**. The environment was thoroughly described in the design chapter. The only elaboration in regards to implementation is handled when describing Rana's structure in a following section (4.4).
- **The API**. Details on how the runtime agent interacts with the predefined static API functions

Following the four sections on the artefact implementation, the implementation details of Rana's structure is described in section 4.4. Finally the im-

plementation details of the multi-threading algorithms are described in section 4.5. All sections of implementation share two secondary goals.

- **Expandability**. It is our intent to lower the complexity threshold for expansion of the API and the user experience. Implementation-wise the user interface framework Qt has minimal coupling with the core and API. So it can be replaced or removed without significant rework of the simulation core.
- **Performance**. Rana is intended to be a high performance framework. Multi-thread support is paramount for maximizing performance on modern systems. Therefore some emphasis has been put on threading agent behaviour.

Small code bits will be shown in some sections to illustrate code-specific functionality.

4.1 The Event

The event has in design been established as the medium for external action representation. In implementation the event is tightly tied to the event handler, which is the entity that defines and handle the events and their references. As the run-time Lua agent is considered the primary target for agent and event design we will both be considering the compile-time and the run-time representations of the event.

4.1.1 Event Representation

An event holds a variable for each of the valid event parameters an agent can define. As the agent distribution mechanism is part of the Core events must transition between the compile-time core and the run-time agent interface, C++ and Lua respectively. So a representation of the event exists on both platforms. Table 4.1 displays the attributes of both event representations.

Descriptor	Lua Type	C++ Type
Propagation Speed	number*	double
Description	string	std:string
Table**	Table	std:string
TargedID	number	unsigned integer
TargetGroup	number	unsigned integer

Table 4.1: Event variables and their corresponding types.
* In LuaJIT 2.0.4 and Lua 5.1 all numbers are represented as reals
** Table is submitted to an internal Lua data serializer (which is part of the Rana utility module) and transmitted to the Rana core as a string. Upon reception of an event the Lua library will de-serialize the table by loading it as a chunk.

Aside from the functional variables the compile-time event also holds a reference counter that denotes how many active event-references that holds a reference to it.

The runtime event handler module is implemented as a Lua module that enables dynamic and safe event generation. From the agent's point of view dynamic event definition is important as the nature of events can be highly variable depending on the type of action that the event represents.

4.1.2 The Event Reference

To tie external events to their relevant agents an event-reference class has been implemented. Whenever an agent emits an event all receiving agents will calculate when that event has propagated to their position. For each calculation an event-reference object will be generated, it holds the following information.

- immutable pointer to the event.
- immutable pointer to the agent that generated the reference.

- Activation time that denotes the take-step phase at which the external event needs to be processed by the agent.

4.1.3 Event Handling

For simplicity's sake we can consider a very simple chorusing simulation. To enable a chorus two different frog behaviours can be defined. One agent acts as an emitter frog that preforms calls at random intervals. The other behaviour is that of a responder agent that generates a special response call the moment it registers an emitter's call.

4.1.3.1 Event Emission

The following two sequences describe the needed steps the system goes through whenever an agent emits an event.

The first details the sequence from emission to submission into the the event queue. A diagram of the sequence is featured in figure 4.1.

- The frog emits an event, containing description "call" and a propagation speed of 343[m/s]. This is done via the event module.
- The event module interfaces with the C++ agent interface and submits the event data along with the id and position data of the agent.
- The core generates a pointer that points to an event object holding the submitted data.
- The core then moves the event pointer to the event handler.
- The event handler then stores the event and marks the next possible handle-event phase as active by submitting it to the time queue.

The second sequence details what happens the following handle-event phase where generation of event-references for all relevant agents are performed. The sequence is featured in figure 4.2.

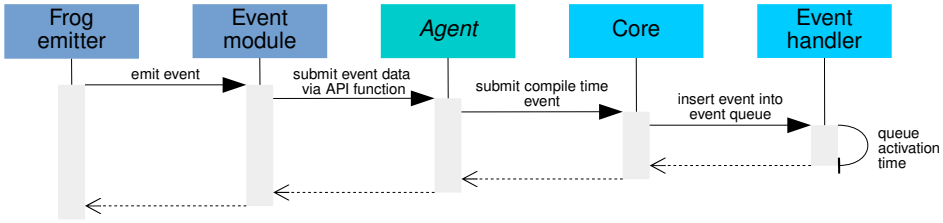


Figure 4.1: Sequence of the event as it travels from the agent, through the API and core and into the event handler

- The agent controller queries the event-handler for the next possible handle-event phase, which is the one following the event submission.
- At the next handle-event phase, the event information will be transmitted to the other active agents which will calculate an event activation time via the event propagation speed. For example, if the receiving agent is 343 meters removed from event origin and the current simulated time is 10[s], the event will arrive at a simulated time of 11[s], as it takes 1 second for the event to propagate to its position.
- The receiving agents will each generate an event-reference to the queued event, which increments the reference counter of the event by 1. So if there are two receivers the event will have a reference count of two.
- The core will move the event references to the event handler which will store them in the event queue's event reference container. The activation simulation time of the events are then submitted to the time queue.

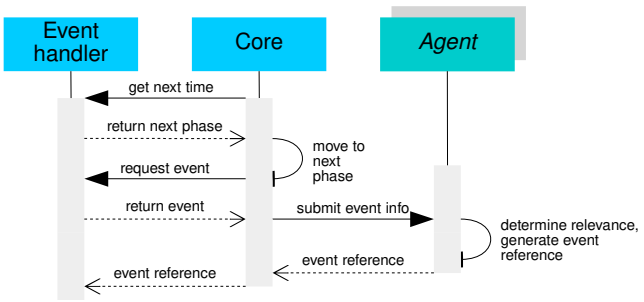


Figure 4.2: Sequence of how the event data is submitted to all agents, which then calculate an event arrival time, and generate an event reference

Listing 4.1 displays the emission of a simple call event. Notice how the event data is passed as a special table containing only the arguments relevant for that specific event. The runtime Lua event is dynamically defined by the agent; the agent designer only needs to define the variables that are relevant for that event and leave the rest as defaults.

```
--include the event module
require Event = require "ranalib_event"
--emit the event
Event.emit{speed=343, description="call"}
```

Listing 4.1: and has description of "call".]Using the event module to emit an event with propagating speed equal to 343[m/s] and has description of "call".

4.1.3.2 Event Distribution

The final part of event distribution is the handling of events which happens when the simulation time is equal to one or more event reference activation times.

The following is an example of how an event reference activation time invokes an agents handle event on the original event data. For a sequence diagram see figure 4.3.

- The core retrieves the event reference from the event handler. The event reference object is moved to the agent interface.
- The interface invokes the agent module, which has a default handle-event function.
- The module de-serializes the event data table and calls the Lua agent's handle event function, if it exists with the reconstructed arguments.
- When the handle event function returns, the event's reference counter is decreased by one and the the event-reference object is deleted from memory as it goes out of scope.

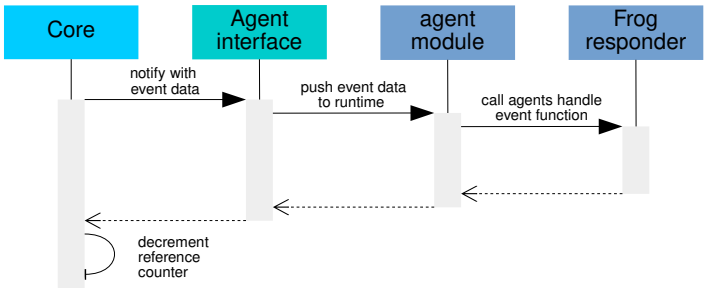


Figure 4.3: The handling of an event reference to distribute event data to a single frog responder agent

There are a number of special cases that can happen during the event handling process.

If the event propagation speed is 0, the event pointed to by the event-reference will be activated on relevant agents in the next available handle-event phase.

If the event propagation speed is bigger than 0, event-reference objects are moved to a container and their activation time will be insertion sorted into the event-handler's time queue

The reference counter of the event is decremented whenever it is activated by receiving agents handle-event function. When the counter hits 0 the event will be moved to a legacy queue, from which it can be written to the hard-drive for later use. This mechanism is used for the event visualizer, which will described in the part following this one on page 141.

Serialization of the Lua table is the most significant performance detriment. A small optimization to event handling can be to submit the table as a string directly, which is entirely possible due to Rana's flexible module design.

Rana's event-queue is an optimized re-implementation of the event-queue from its curses driven precursor. The event queue handles both event-reference objects and events.

4.1.4 Valid Event Table Types

As has been mentioned event submission can be handled dynamically via the event module. The next listing 4.2 shows how two events with similar data can be generated. One is done by submitting a Lua data table while the other defines the data in a concatenated string.

The data table is run through a serializer implemented in a Rana module, while the other is directly translatable to a standard C++ string. Both are translated into a string on agent to simulation core transition.

```
-- generating and submitting a table, depicting the amount of
   calls the agent registers pr. second:
dataTable = { incomingCallPrSecond=incomingCallsPrSecond,
   closestNeighbour=NeighbourID }
Event.emit{description="callData", table=dataTable}

-- generating and submitting a serialized table.
dataStringTable=
    "{incommingCallsPrSecond="..incommingCallsPrSecond.."
       closestNeighbour="..NeighbourID.."}"
Event.emit{description="callData", table=dataStringTable}
```

Listing 4.2: Data event, two types, same result. The event module will detect whether a string or table has been submitted and take proper action.

Performance can always be a concern when doing event design and implementation. Submitting string data rather than tables will speed a simulation up, especially if it is attached to a frequently occurring event.

4.1.5 Conclusion

The event has been established in its two representations. The event handling mechanism has gone through a number of implementation iterations, to optimize performance and memory use. All events are stored in memory via the unique smart pointer paradigm of C++, which incurs a very low overhead over normal pointers and ensures that objects like the event-references are deleted when they are out of scope.

The current event implementation is flexible and provides a powerful interface exchanging inter-agent communication that supports near infinite precision in relation to real-time event propagation.

4.2 The Agent

In the section we introduced the concept of two different agents: an emitting frog agent that could emit a call, and responder agents that can emit a response to that call. Here we will delve into the implementation mechanics that enables their behaviour

The base compile-time agent is implemented as an abstract class that holds the agents defining variables which were listed in the design chapter, such as position and ID.

The class also has abstract definitions for the four simulation-critical functions: initialize agent, handle-event, take-step and clean-up.

4.2.1 The Agent State

Lua as a programming language is designed with C/C++ interfacing in mind. This means that Lua provides an API which allows the C++ program to initiate one or more Lua states which allow it to interface with the runtime Lua program.

To utilize the Lua C++ interfacing functionality for agent design, an agent interface class is implemented. It is derived from the agent class and serves as the link between the compile-time agent and the run-time Lua agent. Each runtime agent instantiation has its own Lua environment, called the Lua State.

The Lua state is a closed environment specific for each agent instantiation. The Lua state is the main Lua coroutine, also called an execution stack. For the purpose of simplicity it can be decomposed into the following parts.

- A general stack that can be used for passing values between the Lua Agent and the Lua agent interface.

- A global space where the Lua agent variables are set and accessed.

- A register for Lua state compatible static C++ functions, that enables the Lua agent to call simulation specific functions.

- A call stack (part of the general stack) that allows for the C++ Agent to invoke Lua functions, such as the ones for handle-event and take-step.

Diagram at 4.4, displays the relation between the Lua agent, the API and the Lua agent interface.

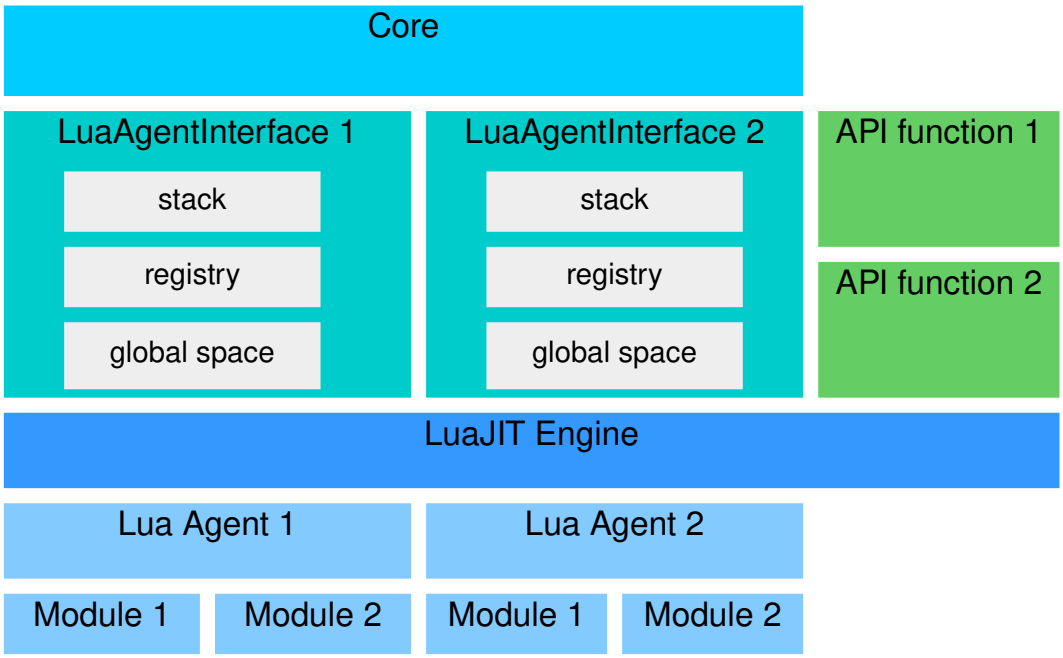


Figure 4.4: Illustration of the Lua engine and agent interface. The simulation core, that instantiates agents. The Lua agent interface, each agent has its own Lua state. API functions implemented as static functions in the Lua agent interface. The luaJIT or Lua engine on which the Lua Agents and modules are run. Lua agents, each Lua agent can instantiate a number of modules, all agents have their own instantiation of modules, and they do not share memory space (atomic).

Lua States are not thread safe, but even though they share the Lua engine each active state can be accessed and their native code will be compiled and processed in parallel.

4.2.2 The Agent Module

Whenever one of the four critical simulation functions is called on the Lua agent interface, it will invoke the default implementation in the agent module. Listing 4.3 shows the default implementation of the takeStep function.

```
function _TakeStep()
    if takeStep ~= nil then
        takeStep()
    else
        -- disable all future calls to the takeStep function.
        StepMultiple = 0
    end
end
```

Listing 4.3: The agent modules default implementation of the take-step function

The agent module exploits the Lua language in that it treats functions as values which allows us to check whether the function exists. This allows the Lua agents to define only functions that are needed for the behaviour they represent.

4.2.3 The Lua Agent

Again we can focus on the simple simulation of the emitter frog and the responder frog behaviour, which allows us to home in on the two simulation critical functions of the run-time agent, handleEvent and takeStep.

Previously, emission of events has been demonstrated but the example omitted the agent specifics. Listing 4.4 shows the full agent behaviour of an

emitter frog agent which emits a call event via the event module.

```
--include the event module
require Event = require "ranalib_event"
require Stat = require "ranalib_statistics"

function takeStep()
    if Stat.getRandom(1,1000) == 1 then
        Event.emit{speed=343, description="call"}
    end
end
```

Listing 4.4: Simple behaviour of an agent that on average emits one event at every 1000 steps. The event is propagating with 343 meters pr. second and has description of "call".

To implement a responder frog agent that responds to the events of the emitter agent we just have to consider the handle event function for the responder agent, the behaviour of a simple responder is implemented in listing 4.5.

```
--include the event module
require Event = require "ranalib_event"

function handleEvent(sourceX, sourceY, originID, description,
   table)
    if description == "call" then
        Event.emit{speed=343, description="response"}
    end
end
```

Listing 4.5: A simple responder agent behaviour

The two types of agents form a very simple type of chorus. The nature of the chorus is simplistic, with the dynamics being determined by the inter-agent distance between the agents. This type of chorus is not a scientifically valid

model; in the demonstration chapter 5 we will detail a more comprehensive example of a chorus simulation.

4.2.4 Movement

When the agent sets the movement variable to true, the following scenario will happen.

- The agent interface will check if the destination variables differ from the current position.
- If they do the a new agent position will be calculated via a set of trigonometric functions; described in equation 4.1 to 4.3.
- If collision detection is enabled via the agent's grid movement variable the collision grid will be updated with the agent's new position only if the new position corresponds to a new collision grid section.
- If the new position overshoots or is equal to the destination, the position is set to be equal to the destination. The overshoot check is direction dependent.

Calculation of new position X and Y via Destination (X_n, Y_n), Speed and Current position (X_p, Y_p). A is the angle

$$A = \text{atan2}(X_d - X_p, X_d - X_p) \tag{4.1}$$

$$X_n = speed \cdot \cos(A) \tag{4.2}$$

$$Y_n = speed \cdot \sin(A) \tag{4.3}$$

4.2.5 Error Handling

Agents are in essence separate programs that interface with each other via the API and event data structures. Writing agent behaviours will be prone to errors and debugging programming errors is potentially quite challenging.

I. Rana

On the agent level Rana handles two types of errors.

- **Parsing Errors**. Whenever the Lua agent initialization function is invoked via the Lua interface, a syntax check is performed. On the first encountered syntax error, an error message along with the path and line number is generated and displayed. The simulation core will then stop the simulation.
- **Exceptions**. If an agent behaviour function crashes, the Lua interface will generate an exception and write out the error message, which is pushed to the Lua stack. The core will then stop the simulation. Exceptions can be caused by the following.
  - Accessing an uninitialized value (nil in the Lua language).
  - Out of stack space, often happens on recursive functions running unchecked.

4.2.6 Conclusion

By adopting Lua as an agent design language, the implementation of behaviour has been simplified. Agent behaviours can also be implemented during runtime. Via the abstract agent class, it is still possible to design compile-time C++ agent behaviours with full API access, which can integrate with runtime agents dynamically.

4.3 The API

The currently implemented API sections are for the most part implemented in pure C++, except for the environment which uses Qt for ease of integration with the Qt user interface and visualization.

The API container and variables and the environment map are reset on subsequent simulations to preserve consistency.

4.3.1 Lua Module Support

All API calls available to the Lua Agent and the Lua Modules are implemented as special static Lua state compatible functions in the Lua interface class. This type of function has a template that requires it to take a single argument, which is a pointer to the Lua state that invokes it. It returns a single integer that denotes the number of return values the function has pushed to the Lua stack. All functions for the runtime API are re-entrant; handling of simultaneous access is separately handled by the API classes.

The API function for retrieval of a random float is displayed in listing 4.6.

```
int AgentLuaInterface::getRandomFloat(lua_State *L)
{
    //load two arguments passed by the Lua Agent from the stack
    double low = lua_tonumber(L,-2);
    double high = lua_tonumber(L, -1);
    //retrieve a double precision random float
    double number = Phys::getMersenneFloat(low,high);
    //push the resulting number to the stack for Lua agent
       retrieval
    lua_pushnumber(L,number);
    //return the amount of results pushed to the Lua stack.
    return 1;
}
```

Listing 4.6: API function for retrieving a random float from Physics

To enable agents to call the API functions, they are registered in the Lua State and will appear to the Lua agent and modules as regular functions. For example, the previously listed random float function has the call signature of *l_getRandomFloat(low,high)*.

4.3.2 Conclusion

The API and modules currently support a number of useful functions for agent implementation. In the demonstration chapter following this one some of the use cases will illustrate some of the module and API functionality, to give a better overview.

In its current state, the number of API implementations is bare-bones, and should be expanded to offer more functionality. It is very expandable and new API sections can easily be implemented by an experienced C++ programmer.

Modules represent a soft target for expansion, as new ones can be added and the existing ones updated without recompiling Rana.

4.4 Structure

As we determined the implementation languages in the design chapter we will tackle the specifics of the implementation of Rana's structure here. This chapter will not describe UI design or general usability: those topics will be covered in the following demonstration chapter.

4.4.1 Interface

All of the classes in the Interface structure have some form of Qt dependency. While the User Interface classes have heavy dependency, the Control and Simulation Output have a lighter dependency. This was done to ease replacement of the user interface as Rana might offer other forms of user interfacing some time in the future.

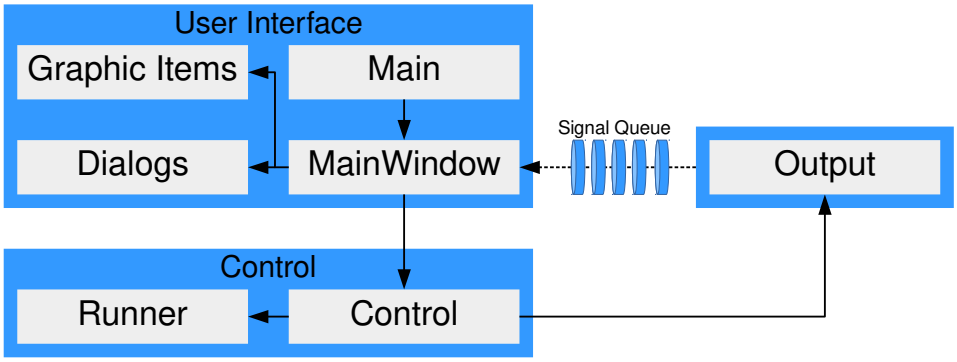


Figure 4.5: Classes and structure of the User Interface, empty boxes are single class sections

Figure 4.5 displays the structure of the interface section of Rana. It has the following classes.

- **User Interface** Contains a set of classes that handles user input and output.
  - **Main**. Instantiates the MainWindow and seeds the random number generator for the API.
  - **Dialogs**. Two dialogues have been implemented: a save event dialogue that allows for saving of events to the hard drive for post-processing in the visualizer and the help dialogue that displays information such as the current version and links to the documentation.
  - **Graphic Agents**. Qt graphic items, each instantiation representing an active agent, for use in live visualization.
  - **MainWindow**. This class encompasses the entirety of Rana's user interface, which is primarily designed in Qt's creators UI designer. The user interface for the simulation part is split into the following panels.
    - **Global**. This is a general purpose panel accessible at all times. It displays progress, and gives the user general control over a simulation.
    - **User Control**. Allows the user to set up a simulation, load a runtime agent, set precision levels and define a map.
    - **Live View**. Supplies a live view of the current simulation, in which the map and agents are displayed as Qt graphic items.
- **Output**. A singleton that allows for dynamic communication between the simulation threads and the MainWindow. Communication is done two different ways depending on the direction.
  - **Atomic Variables**. A number of atomic variables are used to allow the user to interface with ongoing simulations, such as stop an ongoing simulation or set a delay on the agent steps, to help the user to follow an ongoing simulation.

- **Signal Queue**. A series of Qt specific information signals has been implemented to allow agents and other objects to communicate with the user interface. The most significant signals are.
  - Agent Position. Once every simulated second the core will compile a list with all agent positions and transmit them to the MainWindow, which will then update all agent positions on the live map.
  - Write String. Whenever an object or agent wants to write a message, Output will compile a String via a dynamic argument list and emit that as a signal so MainWindow can output it.
  - Remove Graphic Agent. When an agent is removed from an ongoing simulation via the API a signal is sent to ensure that Mainwindow updates its live view accordingly. Contains the ID of the removed agent.
  - Add Graphic Agent. When an agent is added a signal containing positions and ID of the agent is sent.
  - Change Agent Colour contains an RGBA colour value and ID. It allows the MainWindow to change the colour of a graphic agent's representation.

- **Control**. Controls the simulation, ensures correct instantiation of the API and agent domain on subsequent simulations. It also handles the initialization and runtime of active agents. Control operates with two thread types:
  - Simulation initialization function in the core will be executed using Qt's native QtConcurrent::run function. This ensures that the user interface is responsive and outputs agent messages, even with hundreds of agents to initialize.
  - A runtime thread, that has been derived from Qt thread class. This serves as the main simulation thread.

- **Runner**. Inherits from QThread, it initiates simulation runtime after succesfull initialization has been performed. Signals Output when the simulation has stopped.

4.4.2 Simulation Core

The core's main responsibility is simulation flow. Its responsibility is to interface the simulation time and the events with the agents during the runtime. It also controls simulation initialization and finalization. Its structure is displayed in figure 4.6. The core is implemented in pure C++14.

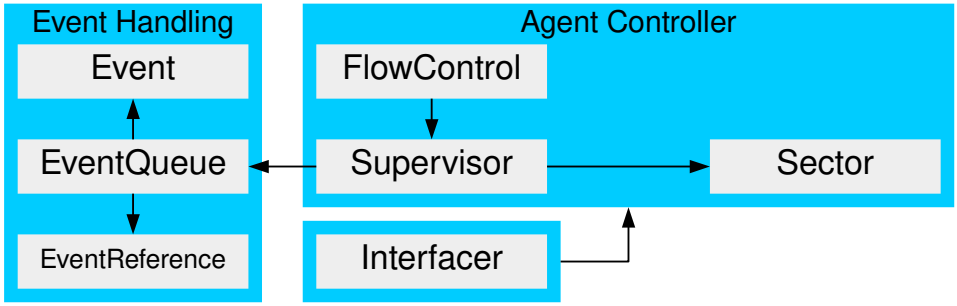


Figure 4.6: Classes and structure of the Core

The core contains the following classes.

- **Agent Controller**. A collection of classes that controls the agents and the simulation phases. It contains the following classes.
  - **FlowControl**. Contains the simulation initialization functions and the runtime logic for retrieving the next phase for the simulation (see section 4.1.3). Notifies the MainWindow with the current progress. Ensures that the simulation is shut down properly on segmentation faults or on stop notifications from Output.
  - **Supervisor**. Controls a number of Sector objects. submits emitted events to the EventQueue and retrieves the next valid handle event phase from the EventQueue at the request of the FlowControl object.

  - **Sector**. Is responsible for a number of agents. Submits emitted events and initiates the relevant phases for each agent.

- **Event Handling**. Handles all event storage and event activation times.
- **Interface**. Handles agent interactions with the core, such as submission of events, adding new agents and changing the colour of the graphic agent.

4.4.3 Agent Domain

The agent domain is responsible for all the active agents in the simulation as well as the modules. Implementation languages are Lua for the Modules and LuaAgent and C++ for the *Agent* and the LuaAgentInterface. Figure 4.7 displays the implementation structure of the Agent Domain.

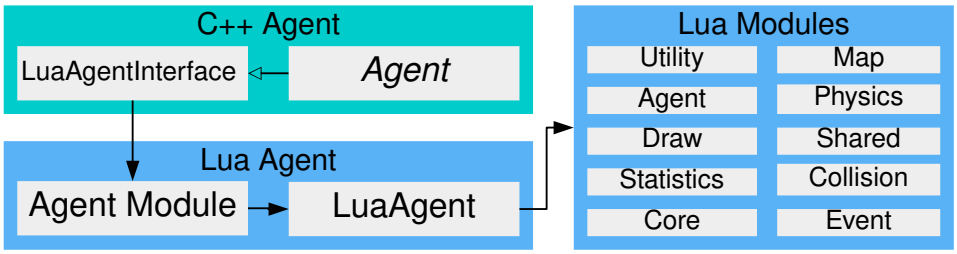


Figure 4.7: Classes and structure of the Agent Domain. ■ Agents implemented in C++. ■ Agents and modules implemented in Lua.

The classes of the Agent Domain are the following.

- **Agent**. The agent is implemented as an abstract class. It defines simulation critical values, such as position and agent ids. It has five abstract functions, the first four are a representation of each simulation phase, the final one being: ProcessEvent which is Used for post-simulation event processing, which will be featured in the part following this one.
- **LuaAgentInterface**. Instantiation of the *Agent*. This links the Lua-implemented agent with the C++ core. More information on this class is in section 4.2.1.

- **Agent module** implements a default function for each of the agent phases of the LuaAgent, which allows agents only to implement phase functions that are relevant to their behaviour.
- **LuaAgent**. This is where the behaviour for each relevant phase is defined.

4.4.4 API

The API consists of a number statically implemented support classes. However, a lot of API functionality is supported directly in the LuaAgentInterface, the reason for this is detailed in section 4.2.1. Each of the interface classes contains a reset function which will reset their containers and variables on subsequent simulation runs.

As can be seen in the structure diagram 4.8, there is very little coupling between the classes as they each have a very specific area of responsibility. This ensures that they can be accessed in parallel.

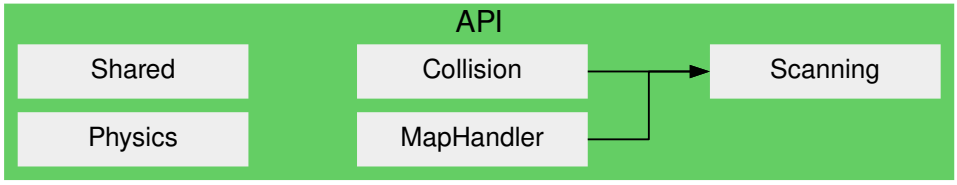


Figure 4.8: Classes and structure of the API.

The API features the following classes.

- **Shared**. Has two maps each indexed by type string. One contains double precision floats. The other contains strings, which are either representations of Lua strings or serialized Lua tables.
- **Physics**. All functions in the Physics are re-entrant utility functions. The random number generator is based on the Mersenne twister implementation in C++11 [cite].
- **Collision**. The collision grid has the following significant artefacts

  - **Grid**. The collision grid is realized as a hash map that stores agent positions, the key is a composite position string equal to $int(x/scale), int(y/scale)$, where x and y are the double precision position of the agent, and scale is the precision level of the grid.
  - **Scale**. Float that denotes the precision level of the grid. $scale = 1$ is a precision level of 1x1[m], $scale = 0.01$ is equal to 1x1[cm].

- **MapHandler**. The MapHandler is the only class with Qt dependency. It employs a bitmap representation of the map via QImage. It holds the functions to set and get individual pixel RGB colour values of the map.

- **Scanning**. Offers a way to generate matrices that are representations of radial masks. It will store previously calculated matrices fast retrieval subsequent radial scans at the same radius. The matrix values are 1 for a valid value and 0 for an invalid value. These masks can be used to simulate agent vision, both for surveying the environment and collision detection (in nature fields of vision is not rendered in squares).

4.5 Threading

Multi-core support poses a couple of challenges. The Lua state is not inherently thread-safe, so agent behaviours should be considered atomic. There is also the sequential nature of Rana's phases to consider.

Multi-core support is achieved by implementation of a master-slave scheme where the simulation Supervisor defines a number of available slave threads, one Sector for each thread. A simulation with a reasonable run-time and precision can have well above a million take-step phases, so it is not prudent to reinitialize a new thread on every take-step phase.

On every simulation initialization, the slave threads are initialized which then yields waiting for a task. Currently only two tasks exist: take-step and clean-up. The clean-up is there to signal thread simulation completion.

Tests have been been performed on threaded event handling, however due to events' sporadic nature it did not provide a performance advantage, even on very event heavy simulations. Therefore only the take-step phase is threaded.

Agents are assigned to sectors via a round robin system. Figure 4.9 displays the logic of both the agent distribution and the take-step phase.

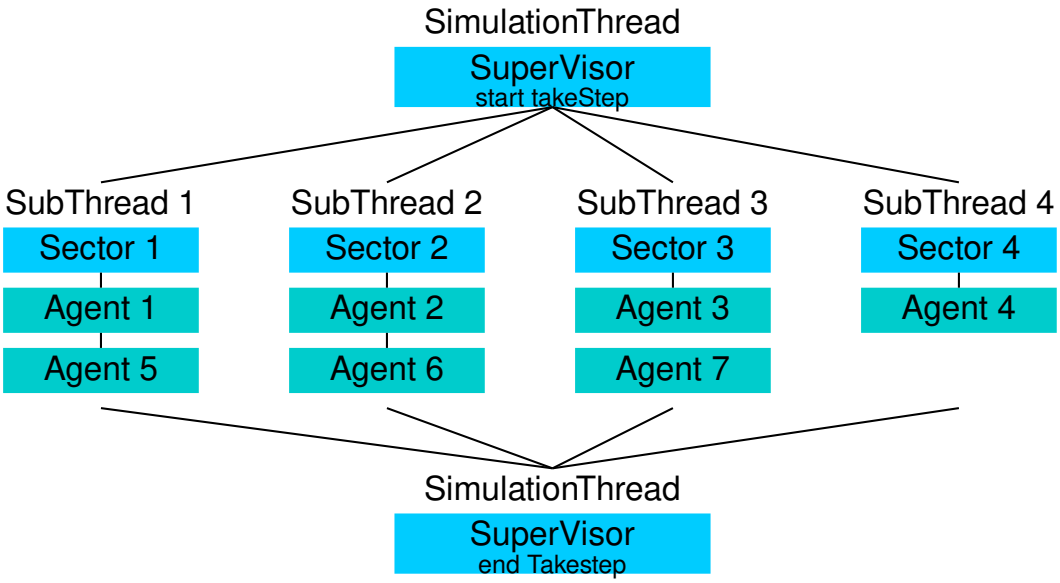


Figure 4.9: Illustration of the take-step threading phase. Beginning of phase the simulation thread signals the sub-threads and yields. The sub-threads then initiate the take-step on the Sector which calls agents' take-step functions sequentially. Once all agents are done the Sector will return and the sub-thread will signal the simulation thread and yield. Once the simulation thread has received a signal for each sub-thread it will move to the next valid phase.

Signals between the simulation thread and sub threads are done via the future multi-threading paradigm. Each sector has a promise (a value another thread can wait for) that is set once the sector is done with the current take-step phase, similarly the Sector has a promise variable with which it defines the next task, this promise is set when the FlowControl notifies Supervisor that the next take-step phase is up.

Condition variables are susceptible to context switches from the operating systems, so sub-threads of the simulation can miss a signal. Testing with condition variables has shown that using condition variables for the task causes

sporadic deadlocks and can make Sectors to miss take-step phases on rare occasions.

Futures are not single shot like condition variables so it can be set while a thread waits for it to be set, or it can be set before the sub-thread has started to wait, the end-effect is the same.

4.5.1 Making API Access Thread Safe

While a lot of the API functions are re-entrant, such as random number generation and simulation time retrieval, other functions require a measure of safety from concurrent read and writes.

All API containers are protected by separate reader/writer mutex schemes implemented via std::shared_timed_mutex from the C++14 standardization (amusingly std::shared_mutex will not show up until C++17). The reader/writer semantics are very safe and due to phases there will never be a situation where there is perpetual readers block a write (unless the agent behaviours access the API in an infinite loop).

The Collision detection module offers a function called *UpdatePositionIfFree* which can ensure atomic movement for the agent only if a position is free, meaning it can read and write a position using the writer mutex.

4.5.2 Concurrency and Behaviours

As agent behaviours are not sequential it is important for the agent designer to note that the level of precision on the take-step phase is at a certain level. For example a precision level of 1[ms] means that within that microsecond the order of agent actions is undefined.

4.6 Conclusion

The current implementation is a complete system that supports dynamic agent behaviours written in run-time. Implementation has successfully been done using only the three decided upon dependencies of Qt, C++ and LuaJIT. This enables us to license Rana as open-source.

In this chapter only some rudimentary agent designs have been demonstrated and the user presentation of the control and visualization itself has not been touched upon. To fully give feel of Rana the demonstration chapter following this will demonstrate the user interface along with a set of agent designs.

Chapter 5

Demonstration

In this chapter we will present the final framework in its current state followed by a set of demonstration models. The agent demonstrations will initially differ from the frog scenario that we have presented throughout design and implementation. The frog scenario served to set up a the system boundary conditions, while the intent of this demonstration chapter is to show some simple models that illustrate various Rana features and useful agent design paradigms.

This demonstration chapter consists of the following sections.

- **User Interface**. A run down of Rana's user interface, which will describe how to set up and start a simulation, control the visualization and interpret the output. Presentation of the user interface is not the most exiting of subjects, and really has no scientific relevance in regards to the subject field. It has therefore purposely been kept to a minimum leaving out all the intricacies and implications of button presses out.

- **Agent Design**. Demonstrates a number of simple agents each showing off various facets of agent design in regards to the four design artefacts

which are events, agents, environment and the API. The following subjects will be demonstrated.

- **Event handling**. A simple ping pong agent behaviour where agents will emit a ping event propagating with the speed of sound. Agents will respond with a pong event on reception of the ping event.
- **Collision detection**. A moving agent that uses collision detection as a repulsion effect. The agents will occasionally scan for collisions within a predefined radius and move in the opposite direction to detected collisions. This results in an even distribution of agents across the map. This simulation has only indirect agent interaction via collision detection; there are no events.
- **Data collection**. Using a simple oscillator agent that serves to illustrate the two pronged approach to Rana data collection namely global and local data collection.
- **Module Agent Design**. A comprehensive demonstration of a frog agent that alternates between a calling and foraging state. The states are implemented as separate modules. Developing advanced agent models is a complex task this final demonstration serves as a venue for easing design by allowing the designer to focus on the agents individual states and their transitions.

5.1 The User Interface

Rana's user interface consists of two main panels, one for simulation control, the second for visualization. It also has some general simulation controls and a progress bar.

The user interface has a general purpose tool bar with the following elements.

- **Command**. Has single exit command to close Rana.
- **Events**. Allows the user to save the most recent simulations events and agent positions via a dialogue window.
- **Options**. The agent has two functions available for outputting html messages, *say* and *shout*. This allows for suppression of the *say* function output, allowing the agent designer to user *say* for debug output and *shout* for simulation critical output.
- **Help**. Opens an 'about' dialogue with links to Rana's source and documentation.

5.1.1 Control

The control panel's main function is to enable configuration of the simulation and textual agent feedback. Agents can relay messages to the user interface via the *say* and *shout* functions. A screen shot with the control panel active is displayed in figure 5.1.

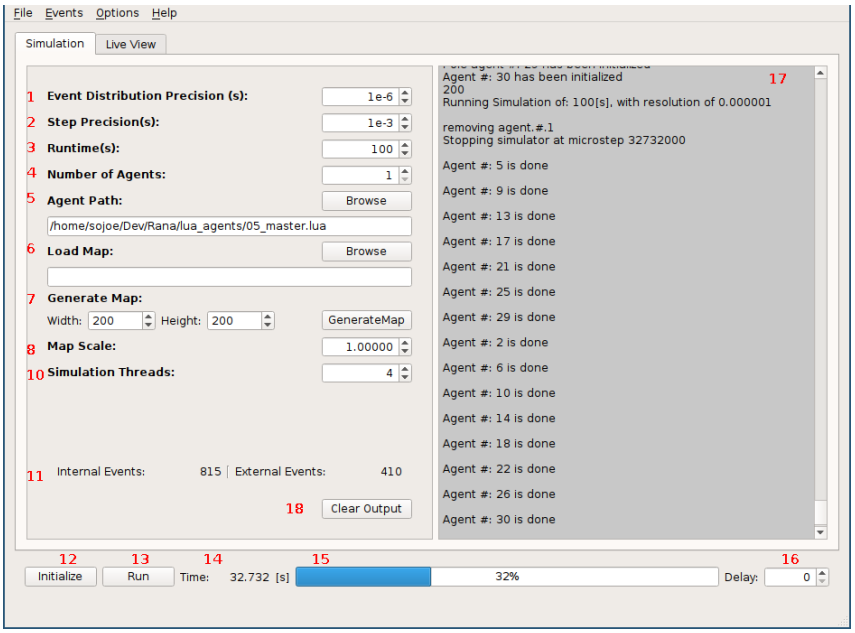


Figure 5.1: The Simulation control interface

Function of the control panel are as follows.

I. Rana

1 **Event Distribution Precision**. The base quantum of time in Rana.

2 **Step Precision**. Precision level of the take step phase, must be equal to or lower than event distribution precision.

3 **Run Time**. Number of seconds to simulate. This means that for a simulation run time of 100 seconds and step precision of 1e-3 we will have $100/1e-3 = 100.000$ take step phases.

4 **Number of Agents**. The initial number of agents to load, of the type defined in the Lua agent file. For multi-agent systems with multiple behaviours you can use a master agent (section 5.2.4).

5 **Agent Path**. Path to the Lua file that defines the agents behaviour either via manual input or the browse button.

6 **Map Path**. Allows for loading a pre-generated map image of type Tiff, Png or Jpeg.

7 **Generate map**. Generates a map with pixel RGB values of 0,0,0.

8 **Map Scale**. Denotes how many square meters each pixel covers, the default is 1.

10 **Simulation Threads**. How many threads that are available for simulation execution.

11 **Event Overview**. Displays the instantaneous number of events and event references.

12 **Initialize**. Executes the initialize function of the agents. Resets all API specific containers.

13 **Run**. Starts the simulation. Changes into a stop button during simulations.

14 **Time**. The current simulation time in seconds.

15 **Progress Bar**. The progress of the currently active simulation.

16 **Delay**. Allows for a delay between take-step phases in milliseconds, slowing the simulation so the user can more easily follow ongoing agent activity.

17 **Output Window**. Displays both agent and system messages such as

simulation status, fault messages or agent output.

18 **Clear Output**. Clears the output window of all data.

5.1.2 Live View

The live view panel gives access to a running simulation and allows users to inspect agents and the map. The view is updated once every 1000 take step phases.

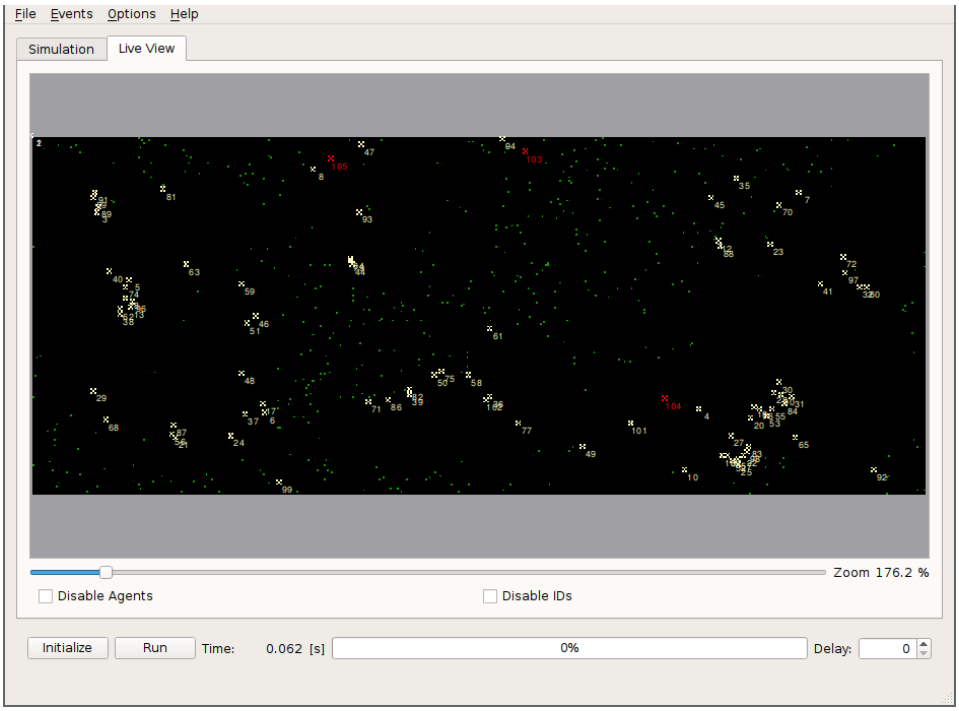


Figure 5.2: The live simulation view

The Live view has the following controls.

- **Agents.** Active agents are rendered as small X's with their ID displayed on the lower right. Agents can be placed and rendered outside the map, this is typically the case for support agents, such as data collectors, so they will not disrupt the visuals.
- **Map.** The map in this scene is black with green pixels, generating by an environment agent (R,G,B value of 0,255,0).

- **Zoom.** It is possible to zoom in on a scene by using the provided slider widget or the mouse wheel. By default, zoom is covering the whole map the zoom level will reset on tab change or main window resizing.
- **Pan.** The user can pan a zoomed-in scene by using the left mouse button.
- **Disable Agents.** The user can disable the display of agents altogether.
- **Disable IDs.** Disables the display of agent ids.

5.1.3 Conclusion

The user interface has gone through numerous iterations via user feedback. In its current state it has been optimized for functionality with as few compromises to user friendliness as possible. While Rana has been tested by students in a practical multi-agent systems course, testing of user friendliness is still a target goal.

5.2 Agents

A number of demonstration agents have been developed to illustrate Rana's approach to modelling. We will start with a very simple model that represents a ping pong agent, which depicts a purely event driven simulation. In the following sections, the agent models will increase in complexity and function. For the sake of consistency, none of the models will use direct API calls, except for the *say* function. It is our recommendation that all API interaction is handled via modules, as they can include call optimizations and error checking (see appendix A.1.1 and A.1.2 on page 217 for examples).

5.2.1 The Ping Pong Agent

The first agent is a very simple model that:

- Initializes at a random position (default model behaviour).
- Emits a ping event at random intervals.

- Emits a pong event if it receives a ping.
- Agents will print out the number of pongs it emitted on simulation exit.

Since this is our initial model we will detail each section that it comprises, such as module inclusion and phase functions.

5.2.1.1 Including Modules

The agent loads up two modules in listing 5.1, an event module that allows for flexible emission of events, and a statistics module that allows for stochastic number generation. The reference to a module is stored in a variable.

```
-- Import Rana lua modules.
Event = require "ranalib_event"
Stat = require "ranalib_statistic"
```

Listing 5.1: Including the modules

5.2.1.2 The Agent's Take Step Phase

For this agent we do not need agent specific initialization. Thus we can leave its implementation out. The syntax is standard Lua and all function definitions and control statements start with the *command* and concludes with *end*.

The agent will emit a ping event at varying intervals. This is done using the statistics module to generate numbers between 1 and $1/Stepresolution$.

The implemented step behaviour is listed in listing 5.2. Note how the STEP_RESOLUTION constant, provided by Rana, can be used to make the emission of events probability independent of simulation step precision. When making real-time critical behaviours this is a very important element to consider if the design is to be tested at varying precision levels.

```
function TakeStep()
    if Stat.randomInteger(1,1/STEP_RESOLUTION) == 1 then
        say("Agent"..ID.." is emiting ping")
        Event.emit{speed=343, description="ping"}
    end
end
```

Listing 5.2: The Pingpong agent's TakeStep function.

5.2.2 Event Handling

The next element is the event handling mechanism of the agent. It will make a decision on how to react to events depending on the incoming event description. If it is a pong the agent will only write out that it got a pong message. Otherwise if it is a ping the agent will emit a pong targeted at the agent it received the ping from. See listing 5.3 for implementation.

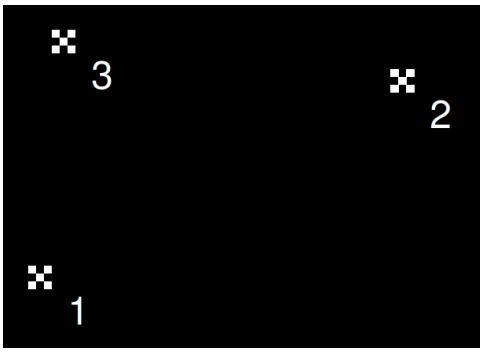
```
function HandleEvent(event)
    -- Check is the event is of type ping, and emit a pong
       response.
    if event.Description == "ping" then
        say("Agent "..ID .." got a ping from: "..event.ID.."
           emitting pong")
        -- Target the source agent of the event.
        Event.emit{targetID=event.ID, speed=343, description="pong
           "}
    -- Check is the event is a pong event, and output a response.
    elseif event.Description == "pong" then
        say("Agent "..ID.." received a pong from agent: "..
           sourceID)
    end
end
```

Listing 5.3: The ping pong agent's Handle Event function. The event data is passed as an event table.

This is the last function definition needed for the agent to provide a fully functioning ping pong simulation.

5.2.2.1 Simulation Output

The simulation output is dependent on the distance and stochastic nature of the event emission. Figure 5.3 shows one possible sample and agent configuration, as captured from the Rana output elements.




```
Agent: 1 is emiting ping
Agent: 3 received a ping from: 1
    emitting pong
Agent: 2 received a ping from: 1
    emitting pong
Agent: 1 received a pong from agent: 3
Agent: 2 is emiting ping
Agent: 1 received a pong from agent: 2
Agent: 3 received a ping from: 2
    emitting pong
Agent: 1 received a ping from: 2
    emitting pong
Agent: 2 received a pong from agent: 3
Agent: 2 received a pong from agent: 1
```

Figure 5.3: Sample output and placement of a ping pong agent simulation. Note how the order of the agents output correspond with the difference in distance between the three agents

5.2.3 Collision Detection

The next model will demonstrate a moving agent that is repulsed when other agents come too near. This agent uses the movement and collision module. The simulation illustrates how agents can interact using secondary mechanics such as collision detection. It has no event activity.

5.2.3.1 Agent Initialization

The agent will start at a random position and set a destination towards the center of the map. The function for agent initialization is listed in listing 5.4.

5.2.3.2 The Take Step Phase

When the agent has reached its first destination the simulation core will set the moving variable to false. When stationary, the agent will do radial collision scans at increasing intervals.

I. Rana

```
function InitializeAgent()
    Move.to{x= ENV_WIDTH/2, y= ENV_HEIGHT/2}
    Speed = 40 -- set the speed to 40[m/s]
    GridMove = true -- enable collision detection
end
```

Listing 5.4: The repulser agents initialization function

If one or more collisions are detected the agent will move away from one of the collisions (chosen at random). This is done by choosing a random valid index within the collision table. It follows the behaviour defined in the state diagram of figure 5.4. States are updated on every take step phase.

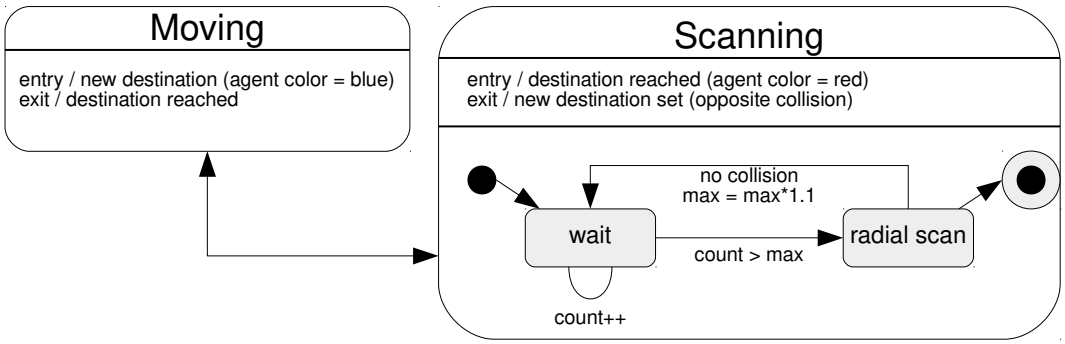
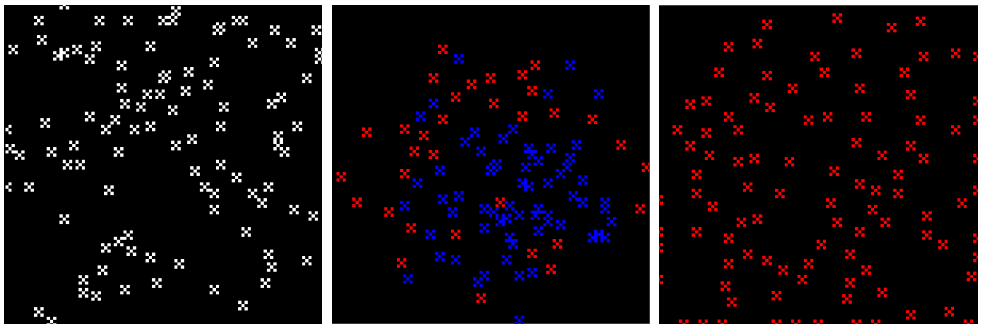


Figure 5.4: The behaviour states of the repulser agents take step phase.

5.2.3.3 Output

There is no textual output for this agent. However, the series of images (figure 5.5) presents the simulation state at various times.

At increasing densities the agents will be pushed towards the edges of the map causing the simulation to be unable to reach a steady state. If steady state is a goal the detection radius should be a dynamic value.



(a) 0 seconds (b) 2.5 seconds (c) 10 seconds

Figure 5.5: Output of a simulation, with 100 repulser agents in a 100x100[m] environment and an agent repulsion radius of 5[m]. All the agents are white upon initialization of the simulation. When in the moving state the agent is blue, in the scanning state they are red.

5.2.4 Data Collection

This simulation will illustrate an approach to data collection which, in our opinion, is a crucial facility to have in place to further the understanding of a multi-agent system. This section features two agents: An active oscillating agent with a stochastically derived period and a supporting data collector agent that gathers simulation wide data.

On every oscillation peak the first agent type will emit an event. Interaction happens when an agent detects an event which will cause it to reset its current oscillation.

This simulation's data collection is two pronged. Firstly, the oscillating agent will collect values at the time of its peak. Secondly the data collecting agent will intercept events and record the number each agent has emitted.

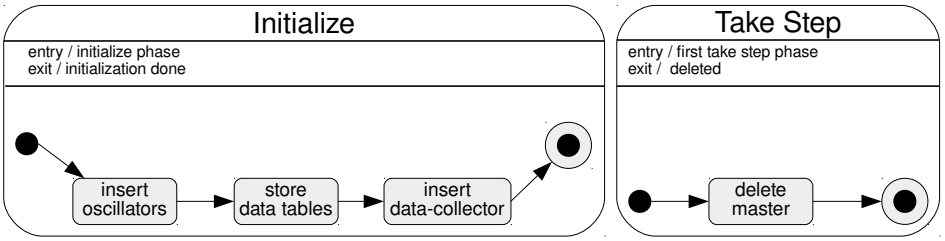
Lastly the data collected will be written out to a comma separated text file using the agent's clean up function, for data processing in a third party tool.

5.2.4.1 Agent Design

As implementation complexity has increased in this section, agent designs are presented using state diagrams rather than source code.

The Master Agent The first agent is a so-called utility agent. It is implemented to serve as an agent specific simulation configurator. Its primary function is to set up the environment, data tables and active agents. This type of agent is crucial to enable complex simulations featuring varying agent populations along with utility agents such as the data collector.

In this simulation the master agent loads and inserts two active agents. The relevant behavioural state diagram can be seen in figure 5.6.



(a) Initialization phase (b) Take step phase

Figure 5.6: States of the relevant phases of the master agent.

The master agent will, on initialization, load up a number of oscillator agents as well as a single data collector agent. It will also initialize two data tables used by the data collector. One is a list containing relevant oscillator agent ids. The other is a data table designed to contain the number of calls each agent has made.

Due to a limitation in the Rana core, agents cannot remove themselves in the initialization phase. Therefore the master will remove itself in the take step phase.

The Data Collector The data collector is yet another support agent. Its task is to collect simulation wide data depicting the number of calls each oscillator has emitted. It will record the amount of incoming events for each agent via the handle event function.

On clean up, the agent will write out the number of events it has intercepted from each oscillator agent to a csv file.

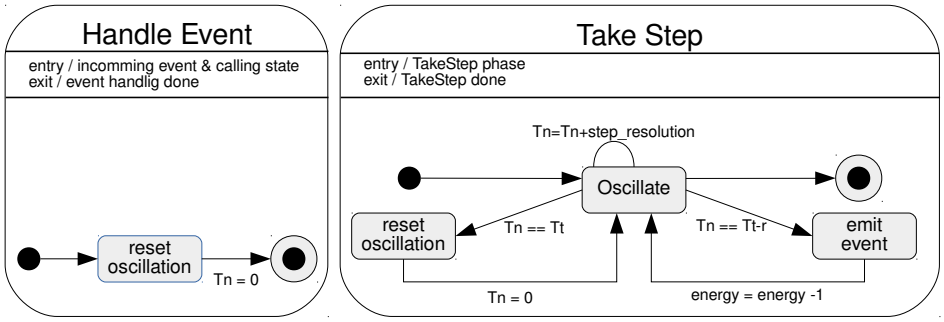
The Oscillator The oscillator agent represents the simulation subject. It is inspired by a scientific model of chorusing males, a model that will explored in detail later in this dissertation. It has the following attributes.

- **T**. Average time period of an uninterrupted oscillation, value is 0.500 seconds.
- **e**. Variance of the time period mean is 0, value is .03 seconds.
- **r**. Fall time of the oscillation 0.100 seconds.

It also has two variable attributes.

- **Tt**. Active uninterrupted period. The peak is at an Tt-r on uninterrupted oscillations.
- **Tn**. Current time of the active period.

The oscillator states in both the handle event and take step phase can be seen in figure 5.7.

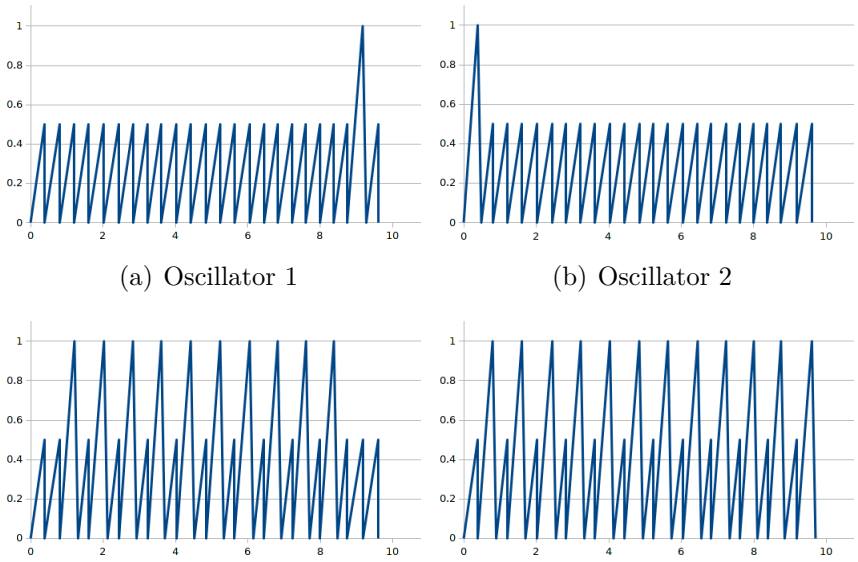


(a) Handle Event phase (b) Take Step phase

Figure 5.7: The main states of the relevant oscillator phases

5.2.4.2 Results

The data collected is a simulation with runtime of 10 seconds featuring 4 oscillators.



(a) Oscillator 1 (b) Oscillator 2

(c) Oscillator 3 (d) Oscillator 4

Figure 5.8: The oscillation values for each individual oscillator in a 10 second simulation, a peak at 1 denotes an event emission, a peak at 0.5 denotes an interruption.

The first set of results is the oscillators' individual performance. The oscillator records a 1 every time it emits an event and a 0.5 on every oscillation reset. Each oscillators output can be seen in figure 5.8.

The short duration of the simulation makes it quite easy to quickly gather how many calls each individual agent has made. However, if we were to introduce 100 agents rather than 4, data analysis complexity would increase tremendously. This is where the global data collector can be a helpful entity to further the understanding of the simulation. In this case it gathers information on the number of calls each individual agent has made throughout the simulation. Data collector output values can be seen in table 5.1.

Agent ID	Number of Calls
2	1
3	1
4	12
5	10

Table 5.1: Number of calls of each agent, corresponding with the number of peaks for each one in figure 5.8

5.3 The Foraging Frog Agent

When designing more complicated agents, the increasing implementation complexity can become difficult to manage, even for the experienced computer scientist. In Rana, it is possible to modularize agent design. This approach allows focus on individual agent states. It also furthers state re-usability for other agent designs.

To illustrate what is basically a composite agent design paradigm, represented by a foraging and calling frog agent will be presented here.

5.3.1 Agent States Via Modules

Modules are not just for providing clean interfaces to the API or for general utility functions. They can also be used as plug-in state representations of an agent. To present the modular agent design. A simple frog model with two sub-states has been developed The two states are:

- **Calling**. During this state the frog is calling to simulate simplified mating behaviour. Each call will consume energy.
- **Foraging**. The agent will forage for food items to charge up the energy needed for calling.

With the general function of the states in place there are also the rules of transition to consider. For this behaviour energy level is set to be the transition

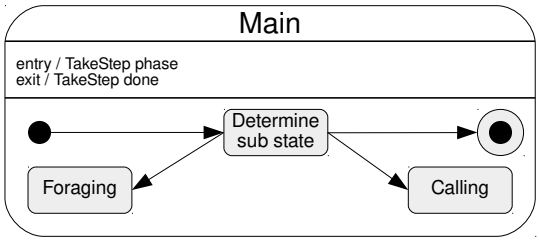
factor between the two states.

5.3.1.1 Frog Agent Design

In agent design the foraging frog agent is detailed. This is done via state diagrams that describes the sub-states and the the state transition rules.

The calling and forage modules each has a specific configuration function. It allows for definition of the states internal variables, such as forage scan range, movement speed and energy.

The agents main state that controls the two sub-states can be seen in figure 5.9.

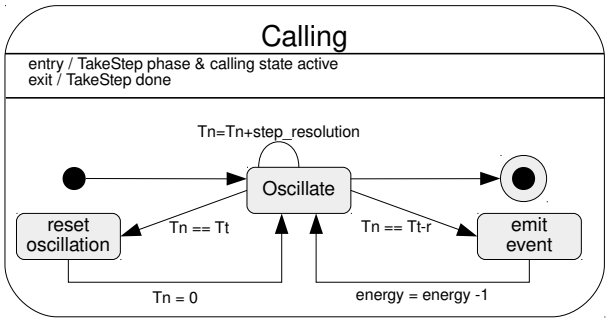


(a) Take Step phase

Figure 5.9: The main state machine of the frog agent. On every take-step phase the frog determines which sub-state it is in. It will transition to the foraging state when energy becomes equal to 0, and transition to the calling state when energy is bigger than the preconfigured limit.

The Calling State To take advantage of an existing agent design the oscillating agent from the previous section has been adopted and adapted to depict a calling behaviour. The only thing that has been added energy usage on calls. The take-step phase can be seen in figure 5.10.

The handle event phase of the oscillator has been disabled as performance of the individual agent is inversely proportional to the number of agents active in the system. This means that this model only calls; it does not react to calls for other agents as the last example did.



(a) Take Step phase

Figure 5.10: The state machine of the frog's calling state, which has been derived from figure 5.7. The only differences between this and the oscillator are the energy usage and entry conditions.

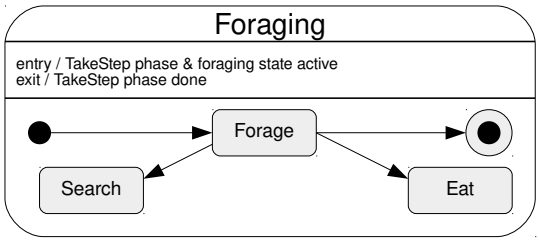
Foraging State The foraging state's goal is to accumulate energy for the agent as it moves around searching for food. When food is located the agent will move towards the food item and attempt to grab it.

The food grabbing process is an example of where the race condition, described in the design chapter (on page 48), can have an effect. If two agents grab at the same food item in the same take-step phase they might both think they got it. This is possible if agent behaviours are executed in separate threads. To prevent it from happening the API has atomic functions in place, that allows for the agents to perform map quantum safe map manipulation by passing two colour values.

The atomic map manipulation function takes the following arguments: an x and y position and two colour values. The first is for checking that the pixel has the colour the agent thinks it has. The other is a new colour of the pixel. If the first colour matches the existing map colour at time of interaction, the function changes the pixel and returns true. Otherwise it returns false. The quantum functions are implemented by using a shared mutex.

There is a similar function in place for collision detection, where the agent only changes position if the collision grid section is free.

The foraging state has two sub-states states, searching and eating. Its state diagram can be seen in figure 5.11.



(a) Take Step phase

Figure 5.11: The agent's foraging state. Only the take-step phase is relevant, as the agent will ignore all incoming call events from calling agents. The agent will transition to the eat state when a food item has been located and back again when the agent has attempted to retrieve the food item. Energy is increased whenever a food item is successfully retrieved.

5.3.1.2 The Food Seed Agent

For the frogs' foraging to function we need to address the representation of the food. To simplify the simulation, food is represented by a specific colour of pixels on the map. To enable this a supporting food seed agent will seed the map on simulation start with a given percentage of food item pixels.

During the take step-phase, the seed agent will check for missing food items and add new ones as required to maintain the preconfigured percentage. To maintain good simulation performance the agent only activates once every 1000 take steps, this is achieved by setting the StepMultiple agent specific variable to 1000.

5.3.1.3 Data Collection Agent

To analyse each agent's performance the data collector of the previous section is added to the simulation. It gathers performance metrics for each agent by counting the number of calls the agent has done.

5.3.2 Experimentation

In experimentation we have set a single driving factor which is the energy level of the agent. The performance or number of total calls is dependent on the

following factors.

- Generation rate of energy when foraging. In the current model this depends on movement speed, amount of energy regenerated on successful forage and density of agents and food items.
- Energy usage per call.

Fixed agent parameters for the two states are listed in table 5.2 for the oscillation state and table 5.3 for the foraging state.

Parameter	**Value**
Time Period	2[s]
Variance	0.2[s]
Fall time	0.5[s]
Call energy cost	1
Energy limit	0

Table 5.2: Oscillation state parameters depicting the nature of the frog agent's calls, when in a calling state, see section 5.2.4 for more information. Energy limit is the value needed for transition into the foraging state.

Parameter	**Value**
Search radius	5[m]
Search move radius	20[m]
Move speed	1,2 or 3[m/s]
Forage energy amount	5
Energy limit	20

Table 5.3: The agent's foraging parameters. Search radius is how far away the agent can detect an food item. Search move radius is how far the agent can potentially move on each new search if no food item is found. Forage amount is the amount of energy each food item provides. The energy limit is the amount of energy needed for the agent to transition into the calling state.

To keep things simple the only variable in this simulation is the movement speed of the agent when it forages. In the simulation we will have three

different species with different movement speeds. The movement speeds are 1,2 and 3[m/s] respectively. Effectively splitting the population up into three different research groups.

Density of food items will be set to 1% which will be maintained by the food seed agent.

The simulation specific values are listed in table 5.4.

Parameter	**Value**
Runtime	3600[s]
Take step precision	1e-3[s]
Handle event precision	1e-6[s]
Map size	200×200[m]
Number of Frog Agents	40×3

Table 5.4: The simulation parameters for the experiment

5.3.2.1 Results

The data collector has written out a set of comma separate text files, one for each population. For rudimentary comparison the results of the simulation has been put into a box plot (figure 5.12). As there is no direct agent interaction the results are quite consistent. While there is an improvement on performance with increasing movement speed, it is worth noting that calling performance does not scale linearly with increasing movement speed values.

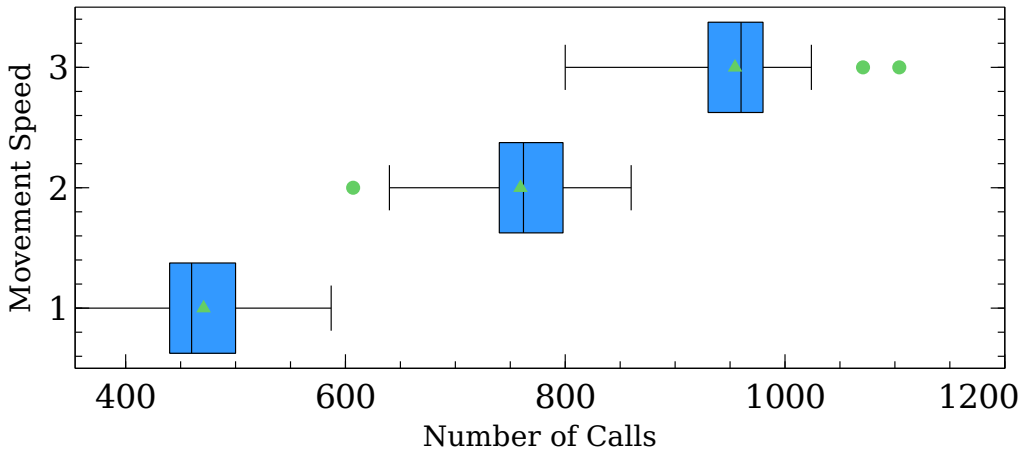


Figure 5.12: Performance box plots (1.5 IQR) for each individual movement speed population, each containing 40 frogs. ● represents outliers. ▲ Represents the mean of each data set.

5.3.3 Conclusion

By using Lua modules to depict separate agent states it is possible to build composite agent behaviours by chaining module states together via rules of transition. In this example we have a multi-layered state machine with very straightforward transitions. The proposition is that we can expand the rules and states by focusing only on the relevant artefacts.

The two featured states are quite simple, but there still are a number of determining factors to consider. To simplify the result presentation we chose to only focus a single variable which is the movement speed but potentially we can have multiple variables. So designing and benchmarking a scientifically sound composite behaviour is a complex endeavour.

5.4 Discussion

When writing a tool intended for use in fields outside computer science user friendliness is an important design goal. In Rana's case usability has to be considered from the perspective of both agent design and the user interface. Even though the task of designing a good user experience is significant, it is a

necessary one.

While general Rana usability has been tested through its use in the SDU multi-agent systems course [15]. Its usability outside the field of computer-science still remains untested. This is particular important to have as it is a long-term goal for Rana in order bridge the gap between computer science and other natural sciences, biology in particular.

Adopting an agent based approach to simulation control and analysis via support agents such as the master and data collector agent, has made it possible to establish a design paradigm that enables advanced simulation and analysis thereof. For an example, we can in theory replicate a real experiment by implementing microphone agents that record agent based events in real-time to see how the artificial world compares to the real one from a researchers perspective.

The modular agent design paradigm, enabled via the Lua modules, is a crucial aspect needed to implement complex agent behaviours. Potentially it allows for experts to collaborate on composite agent designs. For further research it would be prudent to provide a domain specific language covering design states and state transitions, preferably via a graphical interface. This can serve to bring implementation complexity and the learning curve down and thus increase user friendliness in regards to agent design.

5.5 Conclusion

Via the agent examples we have demonstrated a flexible basic agent design paradigm with few constraints. The API and module approach allows for flexible agent design with very few compromises. The biggest constraints are currently the limiting pixel based environment and that agent limited to 2D in both API and visualization.

II

Event processing

"Very well", he told himself, *"so the universe is not quite as you thought it was. You'd better rearrange your beliefs, then. Because you certainly can't rearrange the universe"*

Asimov, Nightfall

Chapter 6

Introduction

We have previously described the event and how it can be used to represent both external agent actions and simulation specific data exchanges for analysis of the simulated multi-agent system.

However, the currently established Rana agent design paradigm has no unified way of processing events based on the events incoming attributes. In nature, or more specifically in a frog chorus the females frogs main task is to perform mating choice. While a few species mating choice, like the poison frog (*Dendrobates pumilio*) [70] is based on visual queues. Mostly mating choice is done via signal processing and localization, which is the case for several species of tree-frog (family *Hylidae*) [23]. When performing mating choices the acoustic driven female frog has to content with being exposed to many different sound-sources, not just for same species males, but also from chorusing of congeneric species, insects and reflections from the environment [18].

Similarly in a simulation in Rana an acoustic driven agent will have to process and evaluate the relevance of incoming events, either to deal with songs from other species or choosing of neighbouring reference callers. For an example, this is the trait of some male species that chorus in triads and duos [83].

II. Event processing

So for an individual animal to function in an acoustic environment they each have to perform physiological processing and filtering of incoming calls. It would therefore be desirable to introduce event-processing functionality to the Rana agent design paradigm. A function that can be used to correspond to the natural animals way of processing incoming events.

Introducing unified event-processing functionality to agent design can also be used to provide an interface for visualizing the event scape. Currently with Rana's live-visualization displaying event activity basically comes down to changing the graphic agent colours or drawing markers on the map. Approaches that makes it hard for a researcher get a tangible overview of agent event dynamics during a simulation. It also makes it hard to illustrate a simulations event dynamics to an audience. As we will see in the following chapters introducing a common function for event-processing also enables us to introduce a method for visualizing the event scape of Rana as a post processing task.

The following chapters will describe the design, development and demonstration of a unified event-processing function and its use for visualizing event activity.

Chapter 7

Design

During the development of Rana we operated with 4 artefacts, the event, the agent, the environment and the API. To enable event-processing and visualization we will expand on the interconnectivity of two of these, the event and the agent. Furthermore we will also establish two new artefacts. They are:

- **The event processor**. As we will se during this chapter the event processor artefact not only provide an interface for designing and realizing event propagation functions. It also provides an interface for event visualization. The event-processing function provides a unified way to design agent behaviours that evaluate events based on the data contained within the events.

- **The event-map**. To enable event visualization we introduce a post-processing visualisation concept, called the event-map. It is a 2-dimensional matrix of graphic objects that displays event intensity values calculated by the event processor at any given simulation time. These values is what enables visualization of the event scape.

As with Rana's simulation design, we will start by describing the two new

artefacts. Once they have been established we will go through the design structure add-ons required to enable event visualization.

7.1 The Event Processor

From literature we know that singing animals usually only use a few fellow callers as reference when timing their calls [83]. When receiving an event the Rana agent can calculate distance to the events source and use that to determine reference callers by sorting them based on distance. However, there is no indication that singing animals gauge neighbouring callers based on distance. Rather it is much more realistic to consider sound intensities and/or frequency as the determining factor [6] [24].

In the Rana demonstration chapter, we have already illustrated the effect high volume simulations can have on the nature of a chorus. Specifically if selection of interrupting sources is not part of the decision process. For example, the number of calls made by the oscillation agent in the Rana demonstration chapter, on page 87, is inversely proportional to the number of fellow oscillators, regardless of inter-agent distances.

When designing a sound-driven simulation to support a frog chorus, we can have sound intensity as a determining factor. Sound intensity is a property of an emitted sound that degrades over distance. Furthermore if we go beyond frog agents and look at hunting bats that emit high frequency hunting calls the sound intensity can have a distinct directional property [71].

As events in Rana are universal they can represent most types of external actions. Some external actions such as a call has some measure of intensity e.g a laser beam has light intensity, emission of sound has sound intensity, and ground tremors can have intensity of force.

To accommodate event-processing the agent design paradigm is expanded with a fifth function, the event-processing function. It offers a unified approach to event-processing for agents that have need of event evaluation, more

advanced than parsing of description and calculation of distance to source.

For example, the frog agent can define a propagation function for calculating an incoming calls sound intensity. When the frog agent registers a call event, it will evaluate that event based on its sound intensity. To simplify the function we can assume that sound propagation is spherical. On reception a frog agent can then calculate the sound intensity level at its position via equation 7.1 and 7.2.

We can define the function to describe sound intensity for spherical sound propagation. r is distance from source, P sound pressure at source. d is a value to prevent numerical singularities, as the denominator can become very small if the inter-agent distances are very small, the value can be set to 0.2821 which corresponds to a denominator value of 1.

$$I_r = \frac{P}{4\pi r^2} \qquad \text{if } r > d \tag{7.1}$$

$$I_r = P \qquad \text{if } r =< d \tag{7.2}$$

By evaluating the sound intensity on event reception the agent can use it to evaluate event relevance. If all call events are emitted containing the sound pressure at source, the agent can sort the relevance of event sources dynamically by evaluating the average intensity of their calls on reception. With frogs this can both be used for female mate selection and for males to determine neighbouring reference callers.

For example, the agent can define a fixed number of fellow agents. If we were to implement such an agent, each agent can potentially determine reference callers by calculating the average call intensity over a period of time. The agent can then use that information to choose which neighbours it will select as references to time its calls.

In short the event-processing functions main purpose is to return an intensity value based on event and position data. Which in turn can be used by the agent to determine event relevance.

7.1.1 Visualizing Intensity

With event-processing established from the behavioural point of view, we can now establish event-processing for use in visualization.

As the event propagation function has been defined to return an intensity level, that level can be visualized via a colour gradient. For the frogs, the event-processing function can be used to generate an intensity level not just at the receiving agent's position but practically at any position in the environment.

The event-processing function is a fifth function which can be called by Rana. However, it is only called during the simulation by the agents on event reception or in a post-processing scenario for visualization.

To visualize events the event-processing function can be used to evaluate event intensities across both space and time.

7.1.2 Conclusion

The event-processing function has been established as a tool for generalizing calculation of event intensity levels. It represents an expansion of the existing agent design paradigm by offering a unified approach to event handling for natural agents. Its functionality is twofold as it is also designed to perform map-wide intensity calculations for the purpose of visualization in the event map artefact described in the following section.

7.2 The Event-map

The simulated environment in Rana is represented by an image, a representation that potentially could be used also to represent processed event intensity levels, each image representing intensities at different time periods. Using images for event visualization is problematic as we would like to enable visualization of various intensity levels across time.

Instead each section of the environment is represented as an object holding an intensity value at different simulation times. As the intensity level represents a z value in the 2 dimensional environment these objects are, in Rana terminology, called z-blocks.

By defining a number of attributes for the z-blocks we can enable adjustment of resolution over both space and time: space by defining how many square meters each z-block will cover and time by defining the number of seconds each time period will cover. This means that the z-block objects covers a number of square meters, and contains an array holding intensity levels at different times.

The event-map is the concept of a scene build by representing an intensity environment consisting of z-blocks. This scene enables the system to display event intensity development at user-defined resolutions.

For example, if we have a simulation map that is 200 by 200 metres and the z-block space resolution is set to 2, we will end up with 100×100 z-blocks each covering 2 square meters. If the time resolution is set to 0.25[s] and we choose to process all events emitted from time 10 to 20[s] each z-block will have an array of $(20 - 10)/0.25 = 40$ intensity values covering 10[s] of simulation time at 0.25[s] increments. An illustration of the construction of the event-map can be seen in figure 7.1.

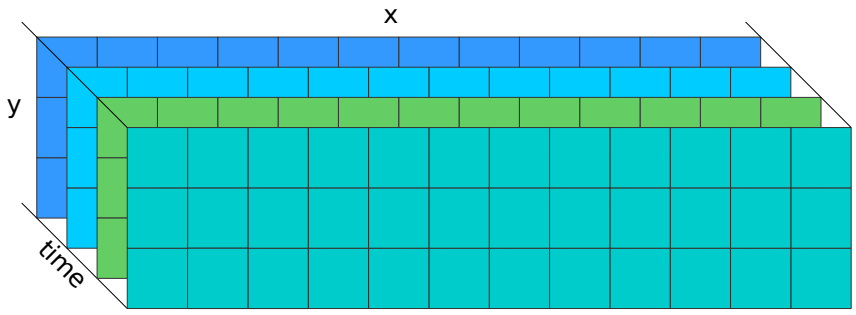


Figure 7.1: Illustration of the event-map, z-blocks are illustrated as coloured squares. Each different colour represents a different period in time.

7.2.1 Intensity Processing

For frogs the event-map's z-blocks can be realized by calculating call events intensity levels starting from the source and then doing a recursive visit of all neighbouring z-blocks until the intensity calculated is below a given percentage of the intensity at source.

Calculation of the sound intensity requires an implementation of the emitting agent's event-processing function. Like the other core agent functions a default function for event-processing is defined in the Lua agent module implementation.

In practice the nearest neighbouring algorithm will start from the z-block covering the source position and calculate an intensity level via the agent's event-processing function. All relevant z-blocks are then visited, and the event-processing function will calculate the intensity level at their position. The event arrival times are registered on each z-block. It depends on following: the duration of the event, the time the event was emitted, the propagation speed of the event and the pre-defined time resolution of the z-blocks.

Usually in nature acoustic events have a duration; for example, the gray treefrog's (*Hyla Versicolor*) call usually consists of a variable number of pulses and can thus have a varied total duration [24]. For example, with a z-block time resolution of $0.25[s]$ and a call duration of 800[ms] the evaluated intensity level of the call will span 4 successive time periods in each evaluated z-block.

To support this the event processor can return a duration as a second value. As the Rana simulation core has no concept of event duration it can be embedded in the event table or for more static durations, in the event-processing function itself.

Because a simulation can comprise many different event types the event-map can sort events via a regular expression engine [74]. Sorting events via regular expressions allows for event-processing based on the description attribute. For example, a frog's call can carry the description of "male_call"

which allows for processing of events with description of "male_call", "call" or even "[a-z]+". Regular expressions provides a flexible tool-set that allows for a great deal of granularity when calculating event intensities.

7.2.2 Saving Events

The processing of events is a post-simulation task. During the simulation emitted events are stored in a temporary container which allows the system to save the event data on simulation completion. This is done by copying event information to a special data event representation along with a string that contains the file name of the agent that emitted the event.

Information required for processing can be contained in the events data table. On processing the only function on the agent that will be invoked is the process event function and the agents position. This is done via a default post-processing agent initialization function. So event propagation should not depend on stochastic agent variables, unless they are stored in the events data table.

In the event save file simulation specific data is also saved. This comprises size and scale of the environment, precision levels, total number of events and the path of the master agent file, which is the data needed to establish the event-map.

Agent position data is another factor to consider. During a simulation the agent positions along with their ids will be cached to a temporary file. When the user saves the events the agent position file will be renamed and copied to the same destination as the event data file. The two files make it possible to process events at various precision levels and visualize both event intensities and agent positions via representation of the event map.

7.2.3 Intensity Representation

Once all the included event intensity levels across the event-map has been processed, it is possible to find the highest and lowest intensity in the predefined time period. These two values can then serve to generate an intensity level reference map that spans across the RGB spectrum. In Rana it spans from blue to violet. Each colour can then illustrate a different level of intensity with very high level of granularity.

Once the event-map has been established the user can browse successive time steps of the z-blocks to see how event intensity develops over time. This approach also makes it possible to play back a simulation and observe agent representations transposed on the event-map.

7.2.4 Conclusion

The event-map is a composite design artefact consisting of a number of sub-artefacts, z-blocks, intensity colour scale and agent graphic representation. Each element has gone through many iterations throughout design. In its current state it succeeds in providing flexible event-processing and visualization. By adjusting the various precision levels and by sorting events appropriately it is possible for a user to generate suitable event maps for observation purposes.

7.3 Design structure

Event visualization is an extension of Rana and so its architecture integrates into the existing design. As agent design in Rana is heavily Lua centric the visualizer is, in its current state, designed towards event-processing using Lua agents only. A design diagram for the visualizer can be seen in figure 7.2.

As the visualizer is integrated into Rana, its functional classes are mostly expansions of pre-existing classes, except for the simulation core that, for illustration purposes has been replaced by a new set of classes comprising the

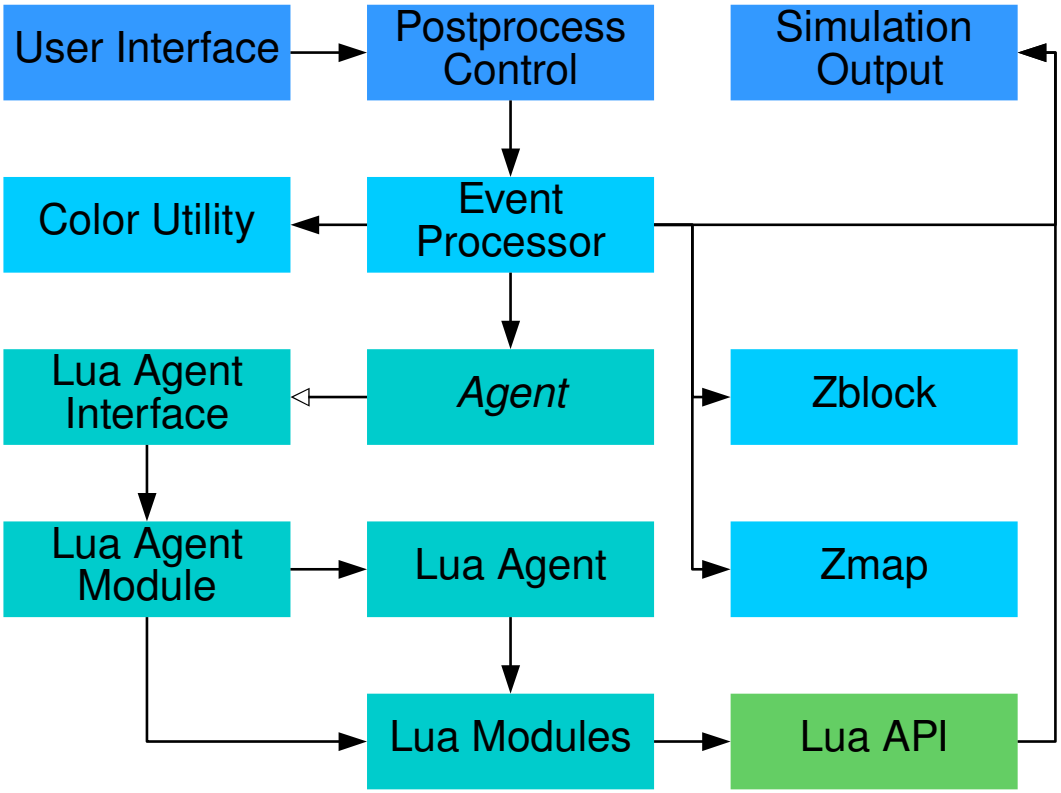


Figure 7.2: Design overview diagram of the visualizer structure. ■ **User Interface**. Handles user interaction and visualization output. It is an extension of the Rana user interface. ■ **The Core**. Handles the processing of events and setting up the event map for user presentation. ■ **Agent Domain**. An extended version of the Rana agent domain to support saving and processing of events. ■ **API**. For the sake of consistency API access is available to agents instanciated by the visualizer

visualization core. Extensions and additions relevant for the visualizer are the following.

- **User Interface**. A post-processing mode has been added to the user interface. This will allow a user to toggle between event visualization and simulation mode in Rana. It has two main panels.
  - **Control**. Allows for the user to load an event save file, define time and space resolution and execute event-processing to generate an event-map consisting of a z-blocks containing all relevant intensity levels for the chosen period.

  - **Output**. Provides event-map visualization when event-processing is done, allows for browsing the various event-map time steps.

- **The Core**. The core handles event-processing it consist of the following classes.

  - **Z-block**. Contains data on the intensity level at the relevant time steps. It will, in its current state store average, cumulative and highest intensity at all timesteps along with event frequency.
  - **Event-map**. The event-map for Rana. It is a 2 dimensional representation of the event environment. It consists of z-blocks and can draw events using the colour utility.
  - **Color Utility**. To transform the intensity levels of the z-blocks to a suitable colour this class will gather the maximum and the mininum intensity levels across all processed events which enables it to convert any intensity value to a suitable colour, spanning from blue to violet.

- **Agent Domain** The agent domain implementations remain largely the same, except for the two new functions have been introduced to the agent class.

  - **Event Processing**: An event-processing function, that can return a duration and an intensity level of an event.
  - Event Initialization: A post-processing initialization function. When the Lua agent is initialized to process an event, this function will be run instead of the regular initialization function. This is done to prevent simulation specific dependencies from disabling the agent, and allow for the agent designer to write event-processing-specific initialization functions. This function can be reimplemented in agent design like the other agent-specific functions.

- **API**. The API remains unchanged. At the start of event-processing all API containers are reset. They will not contain simulation specific data.

7.4 Conclusion

The event visualizer has been designed to provide an easy way for the user to view event activity across a simulation. Events are a significant artefact of the Rana simulation but due to their highly customizable nature the live view does not offer a good interface for visualizing events during runtime. The event visualizer provides that interface.

It expands Rana's current functionality by extending the interface and agent domain and replacing the simulation core with an event-processing core.

The extension of the agent design paradigm to include the event-processing function offers a way for the agent designer to include event propagation processing during a simulation. This means that it is not exclusively a post-processing artefact but it should also be used to design intensity handling of incoming events.

The visualizer is designed to offer a way to make the events more tangible both in simulation design and evaluation. This will be illustrated in the upcoming demonstration chapter on page 127.

Chapter 8

Implementation

This chapter is a description of implementation of the event visualizer. Like the simulation implementation chapter it will start with the the event-processor and the event-map.

Implementation of the visualizer shares the same secondary goals as the simulator itself; performance and expandability. The main difference here is that the visualization core will be heavily tied to the Qt framework with reliance on Qt containers and graphic classes. It is also in a less mature state than Rana's simulation core, which is why the secondary goals hold less prominence in its current state.

As with design the Lua agent is the focus area in this chapter, as the C++ agent in its current state has no implementation supporting event visualization.

8.1 The Event Processor

As was established in design, we want to enable a unified event-processing paradigm that can be used both during simulation and for post-processing visualization. To do this the Lua agent module has been expanded with two new functions, event-initialization and event-processing. The simulation core

has also been expanded to allow for caching of agent positions and saving of simulation and event data.

8.1.0.1 The Event Processing Function

The event-processing function can be used in a simulation by the agent during the event-handle phase to process events in order to determine event relevance. The function returns a calculated intensity value and an event duration. This is also the function that will be called during processing of the event-map. The default implementation of the event-processing function can be seen in listing 8.1.

```
function _ProcessEvent(sourceX, sourceY, posX, posY, time,
    serialTable)
    --check if the agent has an event-processing function
    if ProcessEvent == nil then
        -- return the default intensity and duration
        return 0.5,0
    else
        --load the serialized event table
        loadstring("_eventTable="..serialTable)()
        return ProcessEvent{sourceX=sourceX, sourceY=sourceY, posX
            =posX, posY=posY, time=time, table=_eventTable}
    end
end
```

Listing 8.1: The default event-processing function always returns an intensity of 0.5 and duration of 0

The event-processing function will, similarly to the handle-event function take an event for argument. An important point is that during event-processing for the event-map the event-processing function is invoked for the file name and the agent id that emitted the event, not for receiving agents. So to achieve consistency between post simulation event-processing and live event-processing,

the function should only rely on data contained within the event itself, event-processing functions across all agents in a simulation should, in most cases, have the same implementation, so they process event intensity and duration consistently. For multi-species simulations this can be achieved by adopting the module approach and implementing event-processing functions in a module shared by all agents.

It is also a possibility for different agent types to have different event-processing functions for the same type of events.

This establishes the event-processing function as a tool that allows an agent designer to write custom event-processing functions that can calculate intensity based on event data and displacement relative to the source. The only hard requirement for the event-processing function is that it returns two values, intensity and duration respectively.

It is possible to process and visualize an event duration equal to 0. In that case the event intensity will still be registered to their relevant z-blocks on processing and will span a single time-slot regardless of time resolution.

8.1.0.2 The Event Initialization Function

The second function is the event initialization function. It is a basic function that allows for agent initialization to be separated from a simulation specific dependencies such as, values stored in API containers: values that can cause agent initialization to fail in a post-processing scenario. For example, in the foraging frog simulation from the Rana demonstration chapter, the master agent stores the colour values for the food seed items, which are then retrieved by each frog agent when they are initialized.

The event-initialization function will be called on event-processing, which requires an instance of the agent file that generated it. The default Lua module implementation can be seen in listing 8.2.

Agents are initialized by parsing the agent file name stored in the data event. The system will assume that the path to the file name is equal to the

```
function _EventInitialization(posx, posy)

    if EventInitialization ~= nil then
        EventInitialization(posx, posy)
    else
        PositionX = posx
        PositionY = posy
    end
end
```

Listing 8.2: The default agent initialization function for event-processing

one stored in the simulation information data contained in the event save file. However, it is possible for a user to redefine the path via the user-interface, making the event and position save files portable.

8.1.1 The Data Files

As mentioned in the design, to generate data for event visualization Rana operates with two binary save-files. One contains data event objects, each representing a single event and a single simulation object. The other holds ids' and agent position data at different time-steps.

8.1.1.1 Event Data File

To save all external events to a file, event information data will be parsed and copied into a statically sized object. Table 4.1 an page 53 from the Rana implementation chapter has thus been expanded to give an overview from Lua to Rana core event to data event. For an expanded event data type overview see table 8.1.

The data event is limited to a maximum number of characters for its description, serialized Lua table and agent filename. While the fixed data sizes are fairly generous this is something the agent designer needs to consider if events are to be compatible with data processing. The data types of the data event are displayed in table 8.2.

Attribute	Lua event	Simulation core event	Data event
Propagation Speed	number	double	double
Description	string	std:string	char[2048]
Table	Table	std:string	char[512]
TargedID	number	unsigned integer	unsigned integer
TargetGroup	number	unsigned integer	unsigned integer
Filename	n/a*	n/a*	char[512]

Table 8.1: Event variables and their corresponding types, expanded to include the data event.
* During a simulation agent, file name is not applicable. The data event uses the file name to inform the event processor which agent implementation that generated them.

When saving the events the simulation-specific data is also stored in a single fixed size object. The event data file is stored with a <path>.kas filename. For example, if the user choose path "home/<user>/" and filename as "foo" the file will be saved to "home/<user>/foo.kas".

8.1.1.2 Agent Data

During a simulation the system will cache the positions of all active agents. Caching is done once every simulated second. To prevent excessive memory use the data is streamed to a local temporary file via the generic C++ std::ofstream class during the simulation. Data points of position and ids are stored for each agent in an array indexed by the current simulation time.

When the user decides to store event data, the cached position data will be moved to share the same path as the event data. So if a user chooses the path of the event save file to be "home/<user>/foo.kas" the system will save the position file to "/home/<user>/foo.pos". This save file paradigm enables the visualizer to locate position data based on the path to the event save file alone.

In its current implementation all position data will be stored regardless

II. Event processing

Attribute	Simulation Data(C++)
Number of Events	unsigned integer
Number of Agents	unsigned integer
Time resolution	double
Macro factor	unsigned integer
Number of micro steps*	unsigned 64 bit long
Environment Width(m)	unsigned integer
Environment Height(m)	unsigned integer
Environment resolution**	double

Table 8.2: The attributes of the data event and its fixed sized data types.
* Total number of steps of the simulation which is equal to runtime in seconds divided by the event precision.
** The number of meters each pixel in the environment covers, this is needed as the environment representation is an image. Including it ensures that the event visualizer can render agent positions and events intensities with correct scale.

of whether the agent has moved since the positions was last cached. Ideally, though, the system would only save position of agents whose position has changed since the last check in the buffer file, that is a space optimization that is yet to be implemented.

8.1.2 Conclusion

The event processor is implemented to support a wide range of event types. By reducing the return values to intensity and duration it has been made possible to both support visualization of intensity across time and space and provide a means for the agent to evaluate incoming events, for example it could be calls in an acoustic driven simulation.

8.2 The Event Map

As was established in design, the event-map is a composite construct that is used render event intensity development over time and space. Furthermore, it can also feature agent representations superimposed on its graphical representation. As the event-map has a resolution over both space and time, with every change of either resolution the user needs to generate a new event-map, as the event-map is a representation of a specific event intensity environment with the pre-defined settings.

In implementation the event-map is a Qt graphics scene consists mostly of z-blocks each showing intensity levels at relevant for their location at a given time as a colour. Next to the z-blocks the event-map will display a colour index which shows the corresponding intensity values.

8.2.1 Z-block

The z-block is the class that holds intensity over time at a specific section of the event-map. The z-block is derived from the Qt graphic item class, which allows it to be included in the event-map graphics scene.

The event-map generator will use the agent's event-processing implementation to calculate an event intensity at each z-block, as each represents a section of the environment. So on each intensity calculation the z-block will retrieve the intensity level and the simulation time at which the event intensity is processed.

Each z-block represents a single pixel and they each have an x and y value. If the simulation environment is 200 by 200[m] and the environment resolution is 5[m], the number of z-block comprising the event environment is equal to 40×40.

8.2.2 Event-map Generation

Event file parsing is a two step process.

II. Event processing

When loading an event data file the system will parse the file for general simulation and event data. By parsing the event save-file and retrieving simulation runtime and environment size the user can define the amount of seconds each time step covers and the environment resolution, which determines the number of meters each z-block covers.

Once the configuration options are set the system can begin generating the event-map. First the event-map will be initialized via the following set of actions.

- Parse the event file and load events emitted in the chosen time frame and load them into memory via a linked list.
- Parse the agent position file and load agent positions recorded in the relevant time frame into memory. This is done using a hash-map containing the key of time and a linked list with all currently active agent position objects. This allows the system to render agent positions relative to the active z-block time by using the lower bound lookup paradigm of the standard template library hash map [49].
- Initialize the event map graphics scene.
- Initialize the z-blocks and place them in a graphics scene, using their x and y values.

Having set up the event-map scene and loaded the relevant events and positions into memory the system is ready for processing event intensity levels for all events. For each event the following process will be run.

- Parse the agent directory from the simulation data, or alternatively use a path defined by the user.
- Parse the agent file name from the data event and concatenate that with the directory path to generate an agent path.

- Initialize the agent with id and position contained in the event file. This will represent the state the agent had when it emitted the event.
- From the agent's position the intensity level for the z-block representing that position is calculated, along with the activation time. The z-block position is then pushed to a hash set to prevent revisiting.
- The event-process function is then executed with the position represented by all neighbouring z-blocks recursively. The function is run for all neighbouring z-blocks until the intensity is a pre-defined fraction of the intensity at source, or the boundaries of the environment has been reached.

When all events have been processed the event map scene is ready for browsing. Using the user-interface a user can browse individual time steps or run a continuous slide-show with a predefined pause between displaying timeslots.

8.2.3 Conclusion

In its current state the event-map supports flexible display of event intensity levels via the z-block's four different intensity modes. The event-map is not implemented for performance but rather it is to support the flexible event-processing offered in Rana.

By delegating the rules of visualization to the z-blocks and event-processing to the Lua agent implementation the event-map is a good tool for dynamic event visualization corresponding to the event intensity distribution rules determined by the agent.

8.3 Discussion

Rana's visualization of events, though functional is still very much a work in progress, which means it makes some assumptions.

II. Event processing

The save file system as a bit clumsy in its current iteration. While the visualizer only buffers the position and event data relevant for the chosen simulation period, the data could be streamed into the visualizer as needed instead. And while the position data is streamed to a buffer continuously throughout the simulation event data has not adopted this approach yet.

The end-goal for the save file system is to offer a single time-searchable stream, containing only the relevant data needed to recreate agents and process event intensity levels. Furthermore, since Lua supports loading data strings as Lua code, it could even be an option to include the agent Lua implementations in the data-stream as strings.

It might also seem like we have limited the functionality of the event-processing function by only defining two return values, intensity and duration. However, it is an option to define any number of extra return values as Lua functions are not defined only by name, not arguments and return values. Defining extra return values to the event-processing function will not affect event-map generation functionality.

The event-processing and visualization implementation is not as complete state as the simulator itself. As a consequence It does not have the same optimizations in regards to performance and stability as the Rana simulation core.

While it is a part of Rana, in distribution it has a separate version number and development upstream, within the programs structure there is very little interfacing between the simulation- and visualization core.

8.4 Conclusion

This will allow visualization that corresponds to agent behaviour. For an example if we were to build a bat simulation in which a drone navigates a series of obstacle agents using sound emissions which are then reflected back via an implementation of the event-processing function.

The event visualizer implementation has expanded the Lua agent interface with two new functions, event-processing and event initialization. The implementation blends seamlessly with the implementation of the Lua agent established with Rana.

Chapter 9

Demonstration

This chapter will demonstrate the event-processing and visualization expansion for Rana. As the core aspects of agent design have already been covered by the Rana demonstration chapter, this chapter will primarily deal with event-processing and visualization.

The chapter consists of the following sections.

- **User Interface**. A description and illustration of the post-processing mode for the Rana user interface. The user interface now has two modes, one for simulation and another for performing event visualization. As before, the intricacies of the user interface interaction are left out, as they have no real scientific value.

- **Event Processing**. The event-processing aspect is presented using an expanded version of the oscillator from the Rana demonstration chapter (section 5.2.4 on page 87). We will illustrate how an implemented event-processing function can be used to evaluate sound intensity levels and be used to choose neighbouring event emitting agents that will serve as references for interruptions of the oscillation.

II. Event processing

- **Event Visualization**. To properly illustrate the event visualizer a simple experiment is designed. It involves a drone traversing through an environment with a number of poles. It navigates the scene using sound emissions.

9.1 The User Interface

Rana *either* performs simulation *or* event-visualization-specific tasks, which is why the user interface for visualization is segregated from the simulation user interface.

Visualization mode is enabled by setting the event visualization switch in Rana's general purpose toolbars event menu. The user can switch between simulation and visualization mode seamlessly unless Rana has an active simulation or is in the midst of processing events.

9.1.1 The Event Processing Panel

Upon initiating visualization mode, the simulation control and live visualization panels are hidden and initially replaced by a single event-processing control panel, displayed in figure 9.1.

The functions of the event-processing panel are as follows.

- **Data File:** Browsable path to the event save file, holding the data representation of Rana events and simulation information. The agent position file is assumed to share the same path with a different handle <path>.pos.
- **Agent Path:** Path to the master agent. Each agent relevant for agent processing is assumed to share the same directory path. The filename is stored by each separate data event.
- **Get Event Info:** When the data file path has been set, using the get-event-info button will parse the event data file. Rana will check

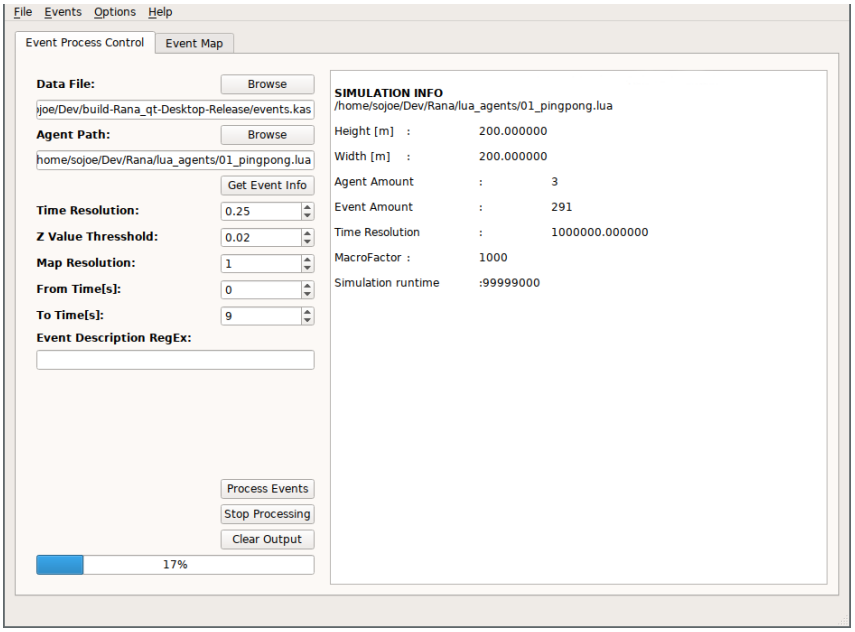


Figure 9.1: The event-processing panel. In this snapshot, data events for the ping-pong agent has been saved to an event file loaded into the visualizer and the simulation data has been retrieved.

data integrity and set the default agent path. Using this button is a prerequisite for performing data processing.

- **Time Resolution:** The time resolution of the z-blocks.
- **Z Value Threshold:** The percentage of event intensity at origin at which the nearest neighbour algorithm will return. At 0 all z-blocks comprising the environment will be visited.
- **Map Resolution:** Number of square metres each z-block will cover.
- **From Time:** The simulation time to start event-processing from. The visualiser will process all events emitted between "from time" and "to time". The indices minimum and maximum values are set when the visualizer parses the event data file.
- **To Time:** see above description.
- **Event Description RegEx:** Used to sort events using Perl regular expression syntax [81]. Only events whose description matches the input regular expression are processed.
- **Process Events:** Processes events using the predefined settings.

II. Event processing

- **Output Window:** Replaces the output window from the simulation window. It presents the simulation data upon pressing the get-event-info button. Furthermore all agent calls to the *say* and *shout* functions performed during event-processing are redirected to this output.

9.1.2 Event-map Panel

On successful event-processing this panel will generate an event-map with a number of user control options for presentation of the intensity levels of the z-blocks that compose the event-map. The panel's functionality is best described through visually, see figure 9.2.

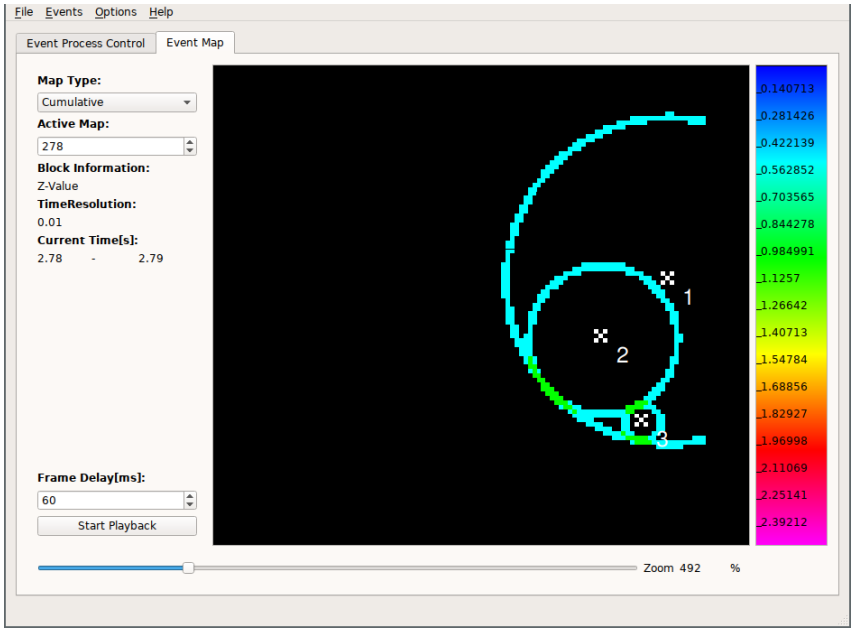


Figure 9.2: The event map panel. The snapshot presents event-processing of a simulation performed with three randomly placed ping-pong agents. The event processor is the default function that will always return an intensity level of 0.5 regardless of z-block displacement from origin.

- **Map Type:** The type of intensity level the z-blocks will display at the currently active timeslot. The different types represent common statistics of interest. Currently the following four types are available.
  - **Additive**. Adds up all intensity levels.

  - **Average**. Takes the average of the intensity levels.
  - **Highest**. The highest of intensity levels.
  - **Frequency**. Number of events active on each z-block.

- **Active Map:** Determines the currently active timeslot for the event map. The number of timeslots is equal to a processed period divided by the time resolution. For example the snapshot in figure 9.2 is generated with a time resolution of 0.01[s] and a period from 0 to 9 seconds, thus it has $10/0.01 = 1000$ displayable intensity environment maps each covering 0.01[s] of simulation time.
- **Block Information:** The user can click on a z-block and this label will appear showing its current intensity level based on map type.
- **Current Time:** The current simulation time presented in the event map.
- **Frame Delay and Start Playback button:** Allows the user to perform playback at varying speeds. Will play back the event intensities by running through the z-blocks timeslots sequentially starting from the currently active map.
- **The Event Map Scene:** Shows the intensity colour values denoted by the z-blocks.
- **Colour Map:** Shows the gradient colour scale and their corresponding intensity values. The scale is dynamically updated when resizing and changing of map types.

9.1.3 Discussion

The visualization user interface has not been exposed to users, as event visualization has not been a part of the MAS summer course or been used for external projects, so it has not undergone the same amount of testing and bug fixing that the Rana simulation interface has received. As a consequence it does not have the same level of completeness as the simulation interface has.

II. Event processing

The incompleteness of the user interface is a problem that comes to light as inconsistencies in presentation of parameters and lack of intensity display options. For example, it should be possible to save current event maps as films for better performance and easier to include in presentations.

9.1.4 Conclusion

The user interface is purely for visualization of events, and while event visualization requires event- processing, event-processing for simulation purposes is independent of it.

In its current iteration, event visualization is functional and despite limitations it offers a number of useful features that makes it a useful feature, as we will see in the following demonstrations.

9.2 Event Processing

The demonstration agent on display here will illustrate the use of the event processor to allow the oscillating agent to process events emitted by its peers. It will use event-processing to determine up to two neighbouring oscillators and use only those for reference when resetting its period. The behaviour is implemented to make the oscillator more robust towards interfering calls, as the original oscillation call frequency suffered with an increase in agent numbers.

9.2.1 Agent Design

The agent inherits the attributes of the oscillating agent described in the Rana section on data collection 5.2.4 on page 87. For good measure the inherited attributes are the following.

- **T**. Average time period of an uninterrupted oscillation, value is 0.500 seconds.

- **e**. Variance of the time period mean is 0, value is .03 seconds.
- **r**. Fall time of the oscillation, value is 0.100 seconds.

It also has two variable attributes.

- **Tt**. Active uninterrupted period. The peak is at $Tt-r$ on uninterrupted oscillations.
- **Tn**. Current time of the active period.

To accommodate reaction to only the two neighbours that have the highest intensity events on reception, the revised oscillator has a couple of new attributes, they are:

- **N1**. A table that holds attributes relevant for neighbour 1. The attributes are.
  - **N1.id**. The id of neighbour 1.
  - **N1.i**. The most recent intensity recorded.
- **N2**. A table that holds attributes relevant for neighbour 2, they are the same as the ones for neigbour 1.
- **P**. Sound pressure at which the agent emits events, value is 1.

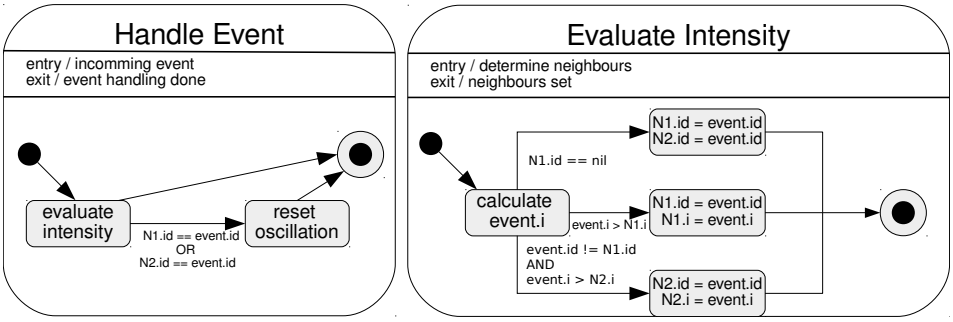
The agent implements an event-processing function, thus overwriting the default one. It will calculate an intensity value (I) by assuming spherical spread using equation 7.2 and 7.1 on page 105 in the implementation chapter.

The agent also inherits the take-step phase state diagram of the oscillator. The only modification is adding the event's sound pressure which is $P = 1$ to the event table when emitting an event.

The handle-event phase has been expanded considerably though as can be seen in the two state diagrams in figure 9.3.

During the handle-event phase the agent will enter an event intensity evaluation phase to evaluate the event's intensity levels. It will update the values stored in N1 and N2 if needed. An important note is that it is possible for

II. Event processing



(a) Handle Event phase (b) The evaluate intensity phase

Figure 9.3: The handle event and its sub-state for evaluating the neighbouring emitters

the agent to have the same reference data in both N1 and N2, this will only happen if the agent only has received events from a single source.

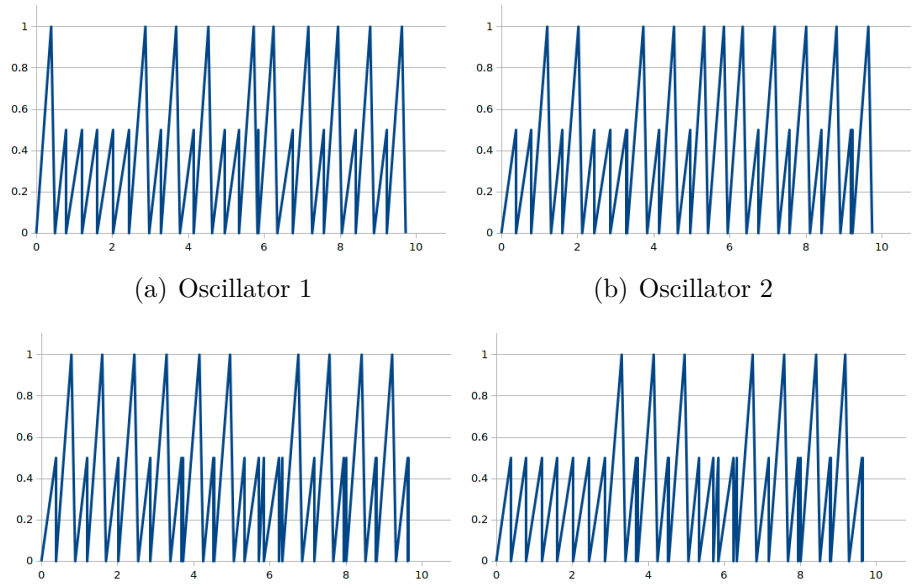
9.2.2 Results

Due to the agent's nearest neighbour algorithm the agent's performance can only be inhibited by two other oscillating agents regardless of agent numbers. Figure 9.4 displays performance graphs for a simulation with four active oscillators. As is clearly seen, each agent performs much better than the oscillator presented in the Rana demonstration chapter.

Again the global data collector has been used. As expected experiments with very high agent numbers (1000) show that each agents average call frequency is consistently the same as long as there are 2 or more possible neighbouring agents.

9.2.3 Conclusion

The modified oscillator agent illustrate how the event-processor can be used to perform event evaluation from the point of view of the agent and during simulation. As the performance of the agents was easily understandable thanks to the data collection scheme in place, we have relegated event visualization to a different simulation in the following section.



(a) Oscillator 1 (b) Oscillator 2

(c) Oscillator 3 (d) Oscillator 4

Figure 9.4: The oscillation values for each individual oscillator in a 10 second simulation. Peak equal to 1 depicts a successful call. 0.5 denotes an interruption.

9.3 Event Visualization

This is a simple simulation to illustrates how the event visualizer can be used to illustrate event activity. The simulation is a an experiment in which a moving bat-like drone navigates towards the sparsest set of two rows of poles using sound-based event emissions. The experiment is inspired by scientific work on bee and bat navigation [67] [17].

9.3.1 Agent Design

The system has two active agents, a single drone and a number of evenly spaced poles. The poles are set up in two rows: a dense set in the upper quarter of the environment, and a less dense set in the lower quarter. The poles are evenly spaced in both rows. The setup can be seen in figure 9.5.

The design of the two active agents is as follows.

- **Pole**. Reflects all sound based events, by emitting an event of lower

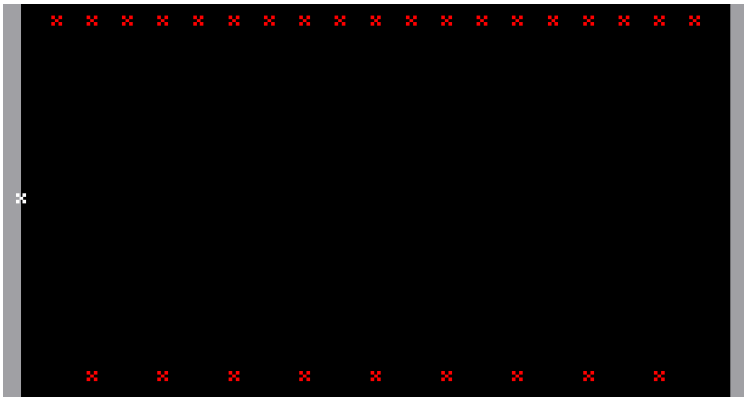


Figure 9.5: The starting setup scenario for the drone experiment. the Red agents are poles. The white agent is the drone. The placement of agents and number of poles in top and bottom is defined by a master agent. The environment is 200 by 200 metres (the image displayed here is cropped).

intensity directed at the event's source. The pole only reflect events emitted by the drone. It has an implementation of the event-processing function that follows equation 9.1 and 9.2 to generate angular determined decay on the emitted events intensity. Event processing requires knowledge of the source positions of the reflected events to generate the vectors needed, that information is embedded in the reflect event.

- **Drone**. The drone starts at position x equal 0 and y equal to half of environment height. At random intervals (once every second on average) it will emit a sound event. It will then intercept sound events via the handle event function. It will evaluate the number of reflected events coming from top and bottom rows. The agent will then move towards the direction with fewer reflections. For example if the density of poles is higher on the top row the drone will move upwards (until a certain threshold is met). Upon reaching the environment boundary the drone will reset its position and start over. The drones sound emission has spherical propagation.

The angular event intensity decay of the reflection is defined by the following equations. First we define angle A that is angular difference between the two vectors. V1 and V2. V1 is the vector from the pole to the source of the reflected event and V2 is the vector from the pole to the processed z-block position

$$A = |\operatorname{atan2}(V2.y, V2.x) - \operatorname{atan2}(V1.y, V1.x)| \tag{9.1}$$

We can then calculate an intensity that decays with increase of angle between the two vectors.

$$I = 1/(1 + angle) \tag{9.2}$$

The drone will steer towards the direction with the most number of reflections. For this experiment it is towards the top of the environment, the drone will never pass the poles. It moves at speed of 10[m/s].

The duration of all emitted events is 0.2 seconds, and they propagate at 343[m/s], for presentation purposes the drones event duration has been shortened to 0.1 seconds.

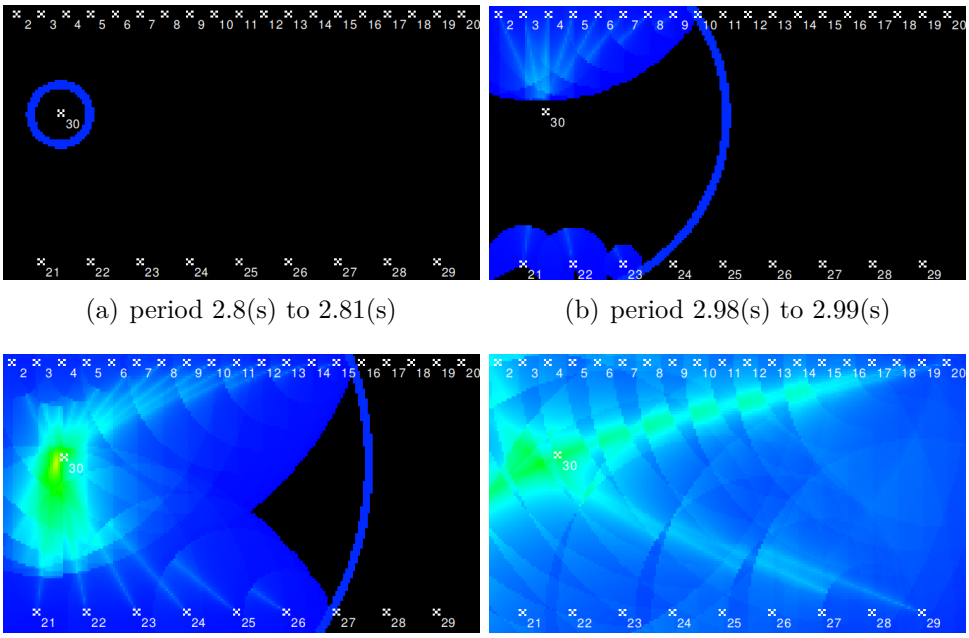
The simulation had a duration of 20[s]. Once the simulation was done all events are saved to the hard-drive, using the event save mechanism of Rana's toolbar.

9.3.2 Event Visualization

Switching to Rana's event visualization mode the event save file is loaded up and the events are processed with a time resolution of 0.01[s] and a 1:1 map resolution.

Two different types of event maps have been snapshot at different timeslots. The additive in figure 9.6 and the average in figure 9.7.

Both figures display the drone's original event as a leading spherical wave with the pole reflection events immediately following as the drone's event impacts the poles. The average displays intensities with better contrast, while the additive display each event's horizon more distinctly.



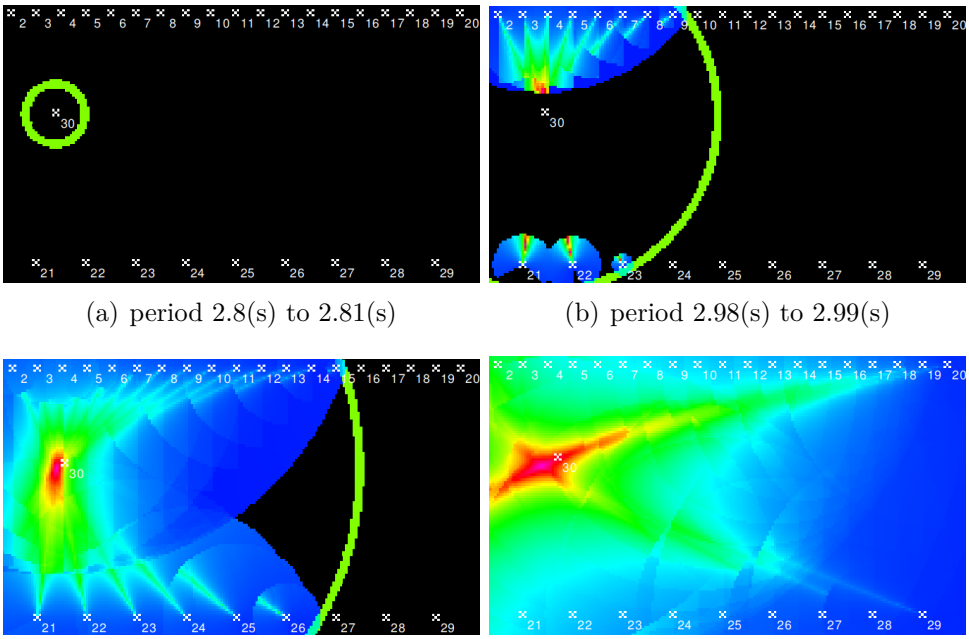
(a) period 2.8(s) to 2.81(s)

(b) period 2.98(s) to 2.99(s)

(c) period 3.12(s) to 3.13(s)

(d) period 3.44(s) to 3.45(s)

Figure 9.6: The visualization of the intensity of events at different times. The setting of the intensity display type is additive.



(a) period 2.8(s) to 2.81(s)

(b) period 2.98(s) to 2.99(s)

(c) period 3.12(s) to 3.13(s)

(d) period 3.44(s) to 3.45(s)

Figure 9.7: The visualization of the intensity of events at different times. The setting of the intensity display type is average.

9.3.3 Conclusion

The drone experiment illustrates the visualizers, and thus the event maps functionality as a presentation tool. It enables visualization of event activity which provides on opportunity to show how the intensity scape develops throughout a simulation.

9.4 Discussion

The Rana event visualization mode is currently in a stable state, meaning it performs uniform generation of event maps and generates consistent output. It is also quite robust towards handling user input errors.

Despite having undergone many iterations throughout development it has yet to undergo user testing scenarios as it has not been part of a the MAS summer course.

The scientific usefulness of visualizing events is debatable, and achieving understandable visualization of a simulations event-scape can be a complex task that requires a good deal of tweaking in the agent event-processing function to provide meaningful event visualization like the one provided by the drone demonstration.

There is also the matter of inter-agent distance. All demonstrations of this chapter exist in environments several hundred square metres in size. If we wish to visualize event intensities in dense frog or insect choruses, it will require a more flexible way of determining the z-blocks environment coverage.

9.5 Conclusion

Event processing represents a strong unified tool for agent-based determination of event relevance. The function complements the four existing agent simulation functions established for the core simulation quite well.

Despite its limitations the event visualizer is a good tool that can illustrate the event scape of a Rana simulation. As shown by the bat drone example it presents a unique view into the world of Rana events. Without event visualisation it would only be possible to show a drone that moved towards the top, again and again without really being able to show the underlying effector, which is the reflected events from the poles.

The event visualizer is a unique concept matching Rana's powerful event artefact and while it is still early in development we believe that, like the event processor complements agent functionality, it will complement the live visualization.

III

Evaluation

"The machine cannot anticipate every problem of importance to humans. It is the difference between serial bits and an unbroken continuum. We have one; machines are confined to the other"

Frank Herbert, God Emperor of Dune

Chapter 10

Introduction

The end-goal for Rana is two-fold, to establish it as an addition to the state of the art for general MAS simulation and to establish it as a scientifically valid tool for implementation of and experimentation on scientific models of acoustically driven systems. To achieve this task the following chapters are presented.

- **State of the Art**. To determine Rana's place in the current state of the art for MAS simulation, a number of existing tools are presented and evaluated against Rana.
- **Simulation of City Traffic**. Rana was used as the simulation and modelling tool to enable simulation on real city sections, to perform experimentation with traffic flow. This chapter will provide a description of the solution offered.
- **Mining Robot Simulations**. Rana is used as course-ware for a computer science course at SDU which tasks students to develop autonomous mining robots. Here the task is described and two of the most interesting solutions are described and benchmarked.

III. Evaluation

- **The Greenfield Model**. To offer credence towards using Rana as a scientific tool for acoustic simulations, a published mathematical model [25] describing male chorusing behaviour has been translated to a corresponding Rana model. The Rana models performance is tested against the results presented in the paper.

Chapter 11

State of the Art Analysis

Rana and its event processing expansion fits into the category of multi-agent systems (MAS) simulation tools. This chapter provide analysis of the current state of the art for MAS simulation to determine where Rana fits in.

We will sample three tools: Repast, MASON and MadKit. Each tool has been chosen for analysis based on the following set of parameters.

- **Popularity**. Each of the reviewed tools has reached a reasonable level of popularity, and has proven itself useful in a varied number of simulation tasks. This also entails that the tools are feature complete and are mature enough to be used by outside parties and not just by the developers of the tool.
- **General Purpose**. The tool can perform a wide range of different simulations. It is not designed to serve a singule purpose only.
- **Availability**. Since Rana is targeted for use in research primarily, it is distributed as open-source and freely available. Likewise the reviewed tools are all freely available to the public and simulations developed using them are publishable using open-source licenses. This also entails that the tools have accessible documentation.

While this analysis is skewed towards the goal of providing the real-time simulation that comprises the Rana simulation. We will (attempt to) evaluate each tool objectively in regards to the general purpose simulations they offer.

In the following section we will evaluate and describe each tool. Once each has been described a general evaluation is performed via a feature table to give an overview of how Rana measures up to each tool.

11.1 Tool Evaluation

To better understand the how each of the tools functions each is described in general terms along with its agent design paradigm. At the end of each section each is evaluated against Rana separately. All three tools all offer platform independent[1] agent design and simulation.

11.1.1 Repast

Repast [53] is a simulation framework that exists in two different versions: a Java-based version for regular users, called Simphony, and an expert C++ version optimised for running simulations on distributed supercomputing systems. For this analysis we will concern ourselves with Simphony as the C++ version is not targeted at regular computing systems and implementation of agents requires expert computer science knowledge which is therefore out of reach for average users.

Repast is probably the most widely-used tool for MAS simulations. It has extensive documentation and has numerous example simulations available, ranging from an implementation of John Conway's game of life [9] to implementations of social networks [52].

The tool is arguably very closely related to another popular tool called Netlogo [76], and at one point in the past Repast even supported agents developed using Netlogo's dynamic agent design language Nlogo. Netlogo has

[1] i.e. support for MacOS, Linux and Windows systems.

been left out of this analysis as it closely resembles Repast and in comparison Repast is better documented and featured in its present state.

11.1.1.1 Agent Design

Repast's agent design paradigm currently supports two well-documented languages for agent implementation.

- **Java**. With Java, Repast offers a number of common infrastructure classes that comprise the Repast simulation. Agents can be designed using inheritance and expansion of these classes.
- **Relogo**. This dynamic language is based on the Logo programming paradigm [57]. Relogo is a concept that allows users with little to no programming knowledge to design simulations. It does this by offering a series of wizards and menus. The implementation language itself is based on Groovy [42], which is an object-oriented dynamic scripting language for the Java platform. By using Relogo, Repast offers a graphic user interface with which the simulation designer can do state-based modelling of agent interactions and transitions rules. Relogo functionality is implemented as an extension to the Eclipse [21] development environment which is required for using Relogo.

Repast Simphony has, at various times, throughout its development history supported a number of design language interfaces such as C#, Python and Nlogo [54]. However in recent years it seems to have deprecated these in favour of Java and Relogo.

11.1.1.2 Evaluation

To make simulation performance independent of CPU execution, times the Repast engine supports moving agent actions forwards in ticks. For complete independence all featured agents must use the tick mechanism provided by the Repast API.

Agent communication is performed using Java method calls, meaning agents call methods on other agents in order to pass information and signal each other.

Visualisation is either implemented separately using Repast's Java-based visualization API or inherited from one of the available demonstration simulations.

When compared to Rana, Repast is a much more mature tool that has proven itself in a wide range of applications.

Repast was part of the existing tool analysis prior to development of the first Rana prototype. At that time it was determined that Repast can be made to support real-time movement and internal agent actions by forcing agents to implement its central tick mechanic. Enabling real-time-bound agent communication and event broadcasting would however require a substantial expansion of Repasts current simulation engine.

Unlike Rana's Lua agent design paradigm, the run-time Relogo implementation is not meant for complex high-performance agent implementations, but rather intended as an introductory language or for rapid prototyping simulations. This leaves Java as the main agent design language. Java being a compiled language ties agent design tightly to the Repast simulation core.

11.1.2 MASON

MASON [44] is a smaller, more lightweight, alternative to Repast. Like Repast it is implemented using the Java framework. MASON sets itself apart by supporting check-pointing, meaning that ongoing simulations can be paused and saved to the hard drive to be resumed later and even on other platforms running MASON.

Originally the goal for MASON was to enable high volume peer-to-peer agent communication and observation thereof. It supports more than a million simple communicating active agents at a time. Despite the tool being optimized for communication it is possible to use it for other types of simula-

tion, and thus it has been used for simulation, ranging from Conway's game of life [9] to simple solar system simulation.

MASON also has a sibling called D-MASON that is designed to perform multi-agent systems simulations on distributed platforms [11].

11.1.2.1 Agent Design

Similarly to Repast, agent design in MASON is done through inheritance of pre-implemented Java agent classes. For visualization both 2D and 3D is featured and is decoupled from agent design.

MASON only supports Java agent implementations. Communication is implemented by calling a messaging system, which allows agents to emit messages to a single target agent. The message contains a method which the receiving agent must invoke to process the message. This sort messaging corresponds to generating a single target Rana event with a propagation speed of 0.

11.1.2.2 Evaluation

Like Repast MASON is also a mature tool when compared to Rana; and like Repast, implementation of real-time propagating event emissions requires an extensive rewrite of the messaging engine and agent synchronization.

Unlike Repast and Rana MASON does not offer any dynamic agent design language. Furthermore the learning threshold is quite steep as there are no ease-in tutorials like the ones offered by Repast.

11.1.3 MadKit

MadKit [26] is the tool that Rana replaced for use in a MAS summer course [15] in 2014 and 2016, which is why it bears comparison with Rana. As is apparently a common theme for MAS simulation tools, MadKit and its agent design paradigm is Java based.

III. Evaluation

It does not have the option for agents to act in ticks like MASON and Repast. Rather agents can feature computing time pauses in the agent code. As a consequence there is no way to perform true-to-real-time simulations in agent behaviours. Furthermore, it means that results are not independent of the performance of platform on which the simulation runs. So inconsistencies can occur especially if a simulation features a high number of active agents .

11.1.3.1 Agent Design

Like both MASON and Repast agents are are designed by extending a pre-existing agent Java class.

Agents can belong to societies, which are designated as communities which contain agent groups. Agents can belong only to a single group. Belonging to a group is a prerequisite for enabling agent communication.

Communication is handled by a central API message system. Agents can register themselves as receivers from fellow group members which can then emit information to them. Agents can only communicate with members of their own group.

11.1.3.2 Evaluation

While MadKit, in this authors opinion, offers a much cleaner compile time agent design interface than either Repast or Mason. It is limited for use as a scientific tool due to being unable to ensure platform independent output.

However for use in general purpose MAS simulation projects in learning MadKit offers a well rounded platform. This has been proven as its use as courseware for the SDU summer course in the years preceding Rana's use.

11.2 Overall Evaluation

To provide a more quantifiable evaluation of each tool's features we have decided on a number of parameters which comprise the real-time simulation

capability that Rana provides. The features are listed in table 11.1.

Feature	Rana	Repast	MASON	MadKit
1. Real-time propagation of messages	x			
2. High agent volume	x	x	x	x
3. Dynamic agent design language	x	x		
4. Platform independent output	x	x	x	x
5. Visualization	x	x	x	

Table 11.1: Feature table for the state of the art frameworks. Xs' marks that a feature is natively available for the framework

A comprehensive explanation of the features and their relation to the individual tools are as follows.

1. **Real-time propagation of messages**. Rana's event engine for distributing events in simulated real-time is a unique feature of Rana. None of the reviewed tools offers a similar feature.

2. **High agent volume**. When measuring agent volume Rana trails behind the other tools. The Rana agent interface has a full Lua state with all API and variables registered and available, this means that the Rana agents take up much more memory that a simple compile time Java agent does. Due to a limitation in the LuaJIT memory allocator Rana is limited to approximately 8000 Lua agents [2], whereas a framework such as MASON can support over 1 million albeit very simple agents. Still any number above 1000 should be considered as high volume.

3. **Dynamic agent design language**. Agent design in Rana is independent Lua scripts. They can be developed and modified during Rana's runtime. This is also an option in Repast with the Relogo agent scripting

[2]Rana can be compiled to use the regular Lua 5.1 engine, which supports a much larger memory space and thus more agents. It is also possible to program agent behaviour in C++ through inheritance of the abstract agent class, but this requires expert programming knowledge.

option. Neither MASON and MadKit has any support for runtime agent modification.

4. **Platform independent output**. Both Repast and MASON support enforced platform-independent output, via a central clock that waits for all agents to perform their ticks, similarly to Rana's take-step phase. None of them have the two-precision scheme for event distribution and internal actions that is a core feature of Rana.

5. **Visualization**. All tools feature some form of visualization. Each requires some re-implementation of existing methods and API calls to offer visualization suited for the simulation at hand. This is similar to Rana's agent and map API functionality. Furthermore both Repast and MASON offer a 3D visualization option.

11.3 Discussion

Rana is built with a very different mindset from the existing MAS simulation tools state of the art. This is evident just by looking at the implementation and agent design languages.

Rana has not been developed to compete with the existing state of the art. Rather it is meant to support simulation of real-time-constrained natural systems, namely animal chorusing, which has been the goal since the development of Rana's precursory prototype [65] began. That Rana in its current iteration can offer general purpose simulations is more of a side-effect due to its flexible design paradigms and it ultimately targeting an ease-of-use design paradigm.

While the state of the art recognizes the importance of agent messaging, Rana takes it one step further and recognizes that messages can in effect be reduced to external agent actions. In Rana we have named that action the Event.

Unlike the state of the art tools, the event concept in Rana takes messaging control from the agent and places it on the message itself. The consequence of this is that if an agent emits a sound all other agents in the simulation can perceive and react to that sound: once emitted its effect is out of the emitting agent's control. While the Rana API makes it possible to emit private and group-wide events, the general paradigm is that events are separate entities perceivable by any agent in the simulation. Another unique feature of Rana is that events can propagate throughout the environment in real-time, with a speed dictated by the event. And while the messages of the MadKit can resemble the instantly propagated targeted Rana event, none of the reviewed frameworks offers anything resembling the event mechanism of Rana.

This brings us to the event visualization feature. This is a unique option that allows for visualization of events. Visualization is dictated by the rules for propagation decided for the event by the emitting agent via the event processing paradigm, similarly to what happens in nature. While MASON and Repast have rudimentary graphic representation of agent peer-to-peer communication none of them offers the ability to define message propagation and visualize how the events propagate and affect the intercepting agents.

While Rana is being used by external projects, time will tell if it can settle within the perceived state of the art for MAS simulation tools. In both execution and simulation design Rana challenges some of the ideas of what comprises a MAS simulation tool today, and given how similar the state of the art tools are to each other we feel that Rana provides the proverbial breath of fresh air to the field.

11.4 Conclusion

Three of the most prominent tools for MAS simulation have been reviewed and evaluated against Rana based on a number of features that are important for providing MAS simulations of both general and real-time-constrained systems.

III. Evaluation

Despite the fact that MAS simulations represent a mature field in computer science this analysis has revealed that Rana has a niche for users looking for something that deviates from the Java-based toolsets and/or having need of real-time critical simulation capability.

Chapter 12

Simulation of City Traffic

In spring of 2014 an M.Sc. project used Rana to perform multi-agent-based simulations of city traffic [66]. The project sought to enable the analysis of traffic on predefined city sections by generating Rana-compatible map data using the OpenStreetMap [27] map service.

Although simulation of city traffic has already been done both in the Netlogo and Repast simulation tools, this project was interesting for two reasons.

- **Realism** The simulation design sought towards optimization of traffic light patterns at various sections of a real city at various traffic loads. Driver agents can move and observe their environment in real-time thanks to Rana's real-time simulation paradigm, which lowers the simulation abstraction level and gives way for better simulation realism.

- **Implementation in Rana**. The project represents a new field for Rana simulation and serves to support the claim that Rana is capable of acting as a general purpose MAS simulation tool.

12.1 Simulation Construction

The task of offering realistic traffic simulations for analysis proved to be a two pronged task. The first was to translate chosen map sections to be compatible with Rana and its agents. The second was the design of the two agent types, the traffic lights and the drivers.

12.1.1 Map Generation

For map generation a separate program, the map generator, was developed in Qt [73] to interface the OpenStreetMap API to Rana. The map generator can retrieve and parse a map section using GPS coordinates and a boundary size.

Upon choosing a map section the generator will translate the map into a bitmap image. Each road sections has a unique colour and the pixel strip width depicts the road type. Aside from intersections the map also supports rendering of roundabouts. See figure 12.1 for an example of a translated map section. The map generator also offers granularity control which allows the user to choose which road types to include.

(a) OpenStreetMap section (b) Generated Rana map

Figure 12.1: An example of a map section that has been converted to the Rana map format.

On simulation start, the map can be loaded into Rana as a map, which can then be populated by relevant agent types.

The map generator will also generate a Lua data file that is employed by a utility agent for placement of traffic lights in the relevant intersections. The file also contains coordinates that define each road, which are used by a second utility agent for placing drivers.

12.1.2 The Agents

To enable the traffic simulation and benchmarking thereof agent a number of different agents has been developed.

12.1.2.1 Functional Agents

The active simulation agents are the drivers and traffic lights.

- **Drivers**: moves along the road with a desired speed and direction. They follow the road using a line-following mechanism. When the driver agent is initialized or if it has reached a destination it will query a navigation utility agent for a new destination and route.

  Drivers can detect other drivers and traffic lights using vision simulated using Rana's collision detection functionality (see appendix section E.1.2 on page 232 for a description of the collision detection module). For dynamic realistic traffic behaviour the driver agent has implemented an intelligent driver model [39], with a number of adjustable behavioural traits. Three different types of drivers where implemented using these traits: aggressive, passive and normal. The traits depict how aggressively the drivers accelerate when stopped at a traffic light or when following other drivers.

- **Traffic Lights**: The traffic light agent controls traffic flow across intersections. It has a duty cycle that determines how long the light will stay green on either side of the intersection. The duty cycle enables the traffic light to have different periods for different directions.

III. Evaluation

When a driver nears an intersection it will query the traffic light for its status. The traffic light will then emit a response telling the driver whether to wait for a green light signal or to continue.

12.1.2.2 Utility Agents

To ensure simulation flow and relevant data collection the following utility agents where implemented.

- **Control**. Configures the simulation by initializing driver agents on free road sections until a pre-configured quota is met. The quota is determined as the percentage of the maximum number of drivers possible on the available road sections.

  It also places traffic light agents in accordance with the map data file.

- **Navigation**. Whenever a driver has been initiated or reached a destination, this agent will generate a new random destination and submit the route to the driver.

  This agent is also responsible for translating road coordinates to road section colours as routes are passed on using colour values to driver agents.

- **Data Collector**. Will at preset times during the simulation emit a "collect all data" event to all driver agents. The driver agents will respond with an event holding a data table which depicts how much time they have spent waiting at a traffic light, since they last responded to a data collection call.

12.2 Results

Several experiments were run on a busy map section taken from the Danish city of Odense. Each experiment was run with two different densities of drivers,

20 and 60%. Three traffic light timings were tested, which were 10, 20 and 30 seconds. The timings determined an unbiased duration of the traffic lights duty cycle. Though these values are hardly realistic the responsiveness of the implemented intelligent driver model proved to be very aggressive and good variance was exhibited across the three different duty cycle timings.

The tests determined that the best timing for ensuring traffic flow for high traffic densities should be a duty cycle of 10 seconds e.g a fast switching traffic light. At the low density, drivers benefited more from longer duty cycles.

12.3 Discussion

At the time of this project Rana was a less mature platform. Some of the features that have been presented in this dissertation were absent: for example, there was no module support, just direct API access. Furthermore, agents could only emit a single event per handle-event or take-step phase. Despite these limitations a flexible traffic simulation was successfully implemented.

The original goal for the project was to try various traffic light tactics at different driver densities to provide optimization suggestions for real life traffic sections. This goal was not entirely realised as only rudimentary experimentation was done within (the rather short) time-frame of the project.

The choice to interface the OpenStreetMap data with Rana's map does represent a missed opportunity. Rather than spending the time developing a piece of conversion software, the time could have been spend enabling support for OpenStreetMaps open OSM format within Rana. Which would have been a good first step away from the limitations of the current environment system.

Despite the limitations posed by an immature platform and the fact that only rudimentary experimentation on the simulation was performed, the project represents a good example of what Rana is capable in the field of general MAS simulation. Furthermore, by coupling the OpenStreetMap service and a Rana simulation the groundwork was laid to perform true-to-real-life simulations

viable for traffic light analysis in various city sections.

12.4 Conclusion

A MAS system for traffic simulation was implemented using Rana as a simulation tool. The project enabled experimentation on both traffic light and driver tactics in a user definable real city sections by exploiting Rana's flexible agent design paradigm.

Chapter 13

Mining Robot Simulations

The MAS mining robot simulation is a mandatory examination subject presented to students attending a 5 ECTS [1] MAS summer-course at the University of Southern Denmark. While the choice of simulation framework was optional, Rana was the officially supported simulation frame work in both 2014 and 2016 (the course only runs once every other year).

13.1 The Simulation

In short, the scenario in the examination subject is this: a spaceship travels to a planet with the intent of mining the ore deposited there. The spaceship carries a base agent that holds a number of explorer and transporter agents. The task for each agent type is the following:

- **Explorer**: roams the planet detecting ore, storing its coordinates which can be transmitted to a Transporter agent.
- **Transporter**: collects ore using coordinates retrieved from an explorer agent and carries it back to base.

[1] In Denmark 1 ECTS is equal to 25 work hours.

- **Base**: a stationary agent that serves as an energy recharge station and ore cache. Bases are responsible for a number of explorers and transporters.

The environment is a torus shaped planet that holds a percentage of ore randomly distributed throughout the environment. The only requirement when implementing agents is that they cannot be purely reactive: each has to have some internal decision logic for communication and its general behaviour.

In addition the simulation has to allow experimentation with the following limiting factors.

- **Energy Level**. Explorers and transporters have a limited amount of energy. All tasks such as movement, scanning for ore and communication cost energy. The agent can recharge its energy level at the base. If it runs out of energy the agent will be unable to take further action and effectively be dead.

- **Communication Range**. Agents have limited communication range for information exchange.

- **Memory Size**. Agents are only capable of holding a limited amount of information, such as coordinates for ore and positions of fellow agents.

This simulation is tick-based rather than real-time. Agents move and communicate in ticks which means that simulation runtime is depicted as ticks rather than seconds, hours or minutes. Rana can emulate the tick-based simulation via its take-step phase; a simulation with a runtime of 10 seconds and a step precision level of 1[ms] equates to 10,000 simulation ticks.

Aside from designing the agents, the system that comprises the agents was to be categorized using the VOWELS paradigm [12]. This paradigm establishes four basic bricks to determine whether MAS core dynamic is primarily agent, environment, interaction or organisation based.

13.2 Results

Although all course participants used Rana for the simulation, there was great agent design variance. Here we have chosen two of the most interesting designs, a distributed blackboard based model [33] and a highly efficient design, designated the dynamic model, that optimizes agent performance using repulsion and information relaying.

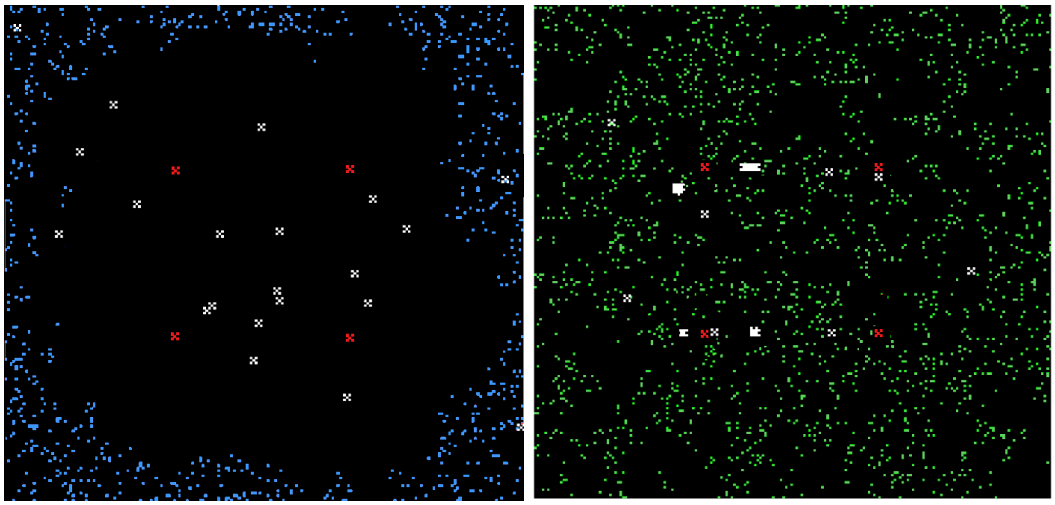
Both models interpret a specific colour in the Rana map as ore. The ore is distributed randomly as separate pixels until ore covers 7% of the map. The map size is 200 by 200 pixels; for this simulation we do not operate in real floating point distances, but instead the map is a grid where each pixel is a possible agent location. Collision detection must be implement so that no two agents occupy the same pixel at any given time.

Data on ore collected was performed by a single passive data collector agent. Each base transmit a completion event to the data collector at the tick at which it was filled.

A simulation with four bases placed in a square formation in the middle of the map was defined, and the emergent mining patterns were observed using Rana's live view, see figure 13.1 for an example.

It might not seem like it but both systems had mined the same amount of ore when the snapshot was taken. The blackboard model exhibits more sporadic mining behaviour whereas the dynamic model uses repulsion to move in an optimized search pattern for the explorers.

To evaluate the performance of each model, 25 experiments were run with the 4 equally spaced bases and the goal of gathering 200 ore in each base. The bases were allowed to cooperate so agents could unload ore and recharge on the nearest base regardless of where they started out from. Different configurations and explorer/transporter ratios were experimented on and the most best performing of both compared using similar communication ranges and energy levels. See figure 13.2.



(a) Dynamic model ■ is ore (b) Distributed blackboard model ■ is ore

Figure 13.1: The mining patterns exhibited with four bases by the two different implementations. The white markers are transporters and explorers and the red markers are bases. The number of active agents is the same for both simulations, but on simulation end they have to move back to the base if they have energy enough. There is a total of 24 transporters and 48 explorers active on each map.

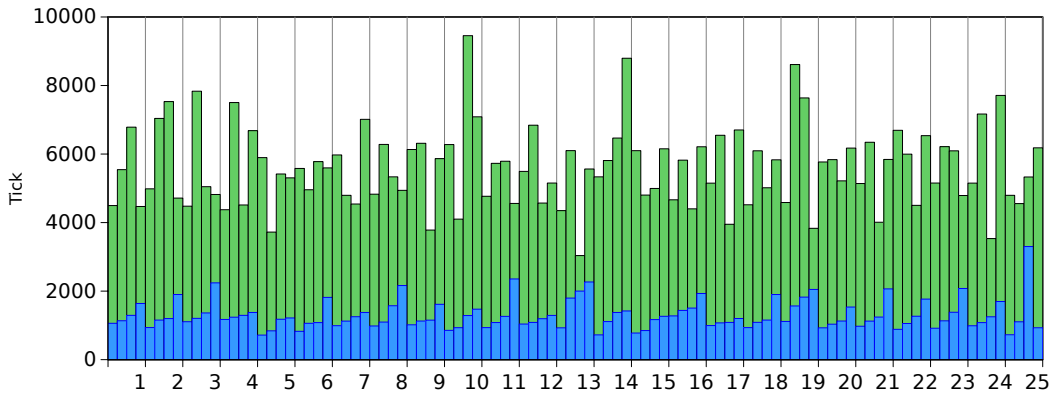


Figure 13.2: The mining performance of each of the two models. ■ is the distributed blackboard model. ■ is the dynamic model. Each column displays the amount of simulation ticks it takes to fill a base with 200 ore. The labels on the x axis each represent a different experiment.

Despite the substantial variance in the experiment results, the dynamic model is clearly more efficient than the distributed blackboard model.

13.3 Discussion

The task of designing and implementing mining robots in a MAS is a good way for students to get a working understanding of multi-agent systems. As can clearly be seen in the experiment presented here, there is a good deal of variance in the emergent properties of the systems. This variance is a very important aspect to consider when working with MAS, both in terms of simulations and real-life systems.

One of the major challenges when working with MAS is the unpredictable patterns that can emerge [16]. For experiments such as these we can control the number of variables within the agent. However, given the autonomous behaviour that was a requirement for this assignment each agent has a number of outside factors that can affect its behaviour. These factors are imposed by the environment and fellow agents. For example, some students' implementations could, on rare occasions, run into mass agent death due to deterministic collision detection algorithms.

13.4 Conclusion

Even though Rana was a relatively immature platform in its first year as course ware, all students attending the course were able to design and implement valid and, in most cases, interesting solutions that solved the assignment. The use of Rana in university level courses further establishes Rana as a tool that is capable of more general purpose MAS simulation tasks.

Chapter 14

The Greenfield Model

In his paper *Precedence effects and the Evolution of chorusing* [25] Michael Greenfield, develops a model of male chorusing behaviour in singing insects. The model enables the exploration of acoustically-driven sexual selection based on the individual chorusing males' ability take precedence over other calling males. This entails emission of calls with a small lead on the calls of fellow males. The reason why the male behaves in such a way is due to a precedence effect in the females mate selection behaviour: the female will turn towards the perceived leading male in a chorus and single him out as a preferred mating partner. Taking precedence can therefore help an individual male attract more females.

The model is relatively simple and effective, and most likely the best one yet to describe the internal workings of a chorusing male in the fields both of insects and of anurans (frogs and toads). It is therefore interesting to translate this model into a Rana agent model in an attempt offer a more general approach. To verify the agent model, it has to be able to reproduce the results achieved by the original implementation presented in the paper [25]. Offering an agent design will provide the option to later expand it using the flexibility of the agent design paradigm offered by Rana.

III. Evaluation

In order to ensure that we do not stray from the source material, all variables and constants, unless explicitly stated otherwise, are drawn from the data presented in the paper.

14.1 The Model

The complete definition of the model is presented in equation 14.1. It takes into account the natural phenomenon of signal propagation delay exhibited by the acoustically driven singing animal. The model is basically an adaptive oscillator. Its period is either lengthened or shortened by incoming signals depending on whether they are received on the downward or the upward flank of the oscillation. To control the rate at which the oscillation period changes a phase response curve value (PRC) is introduced.

$$T_t = s \times [(d + l/v) - (r - t)] + [(T + \epsilon) + (y - x)] \tag{14.1}$$

The mathematical model that controls the oscillation for the male caller.

$\mathbf{T_t}$ is the total duration of a complete oscillation period it depends on the following variables:

- **T**. Average time period of an uninterrupted oscillation. Value is 0.500 second.
- ϵ, Variance of the time period. Stochastically generated on each new oscillation with a value between -0.03 to 0.03 second.
- **r**. The oscillations total falltime, denotes the length of the downward flank of the oscillations. Value is 0.100 second.
- **t**. Interval from peak to call. The model emits its call on the last part of the downward flank of the oscillation. value is 0.060, this means that the call duration is equal to $0.100 - 0.060 = 0.040$ second.

- **x**. Duration of the call. As determined above that value is equal to 0.040 second.
- **l**. Is the distance to an intercepted call in metres.
- **v**. Is velocity of the call. Value is 343 [m/s].
- **y**. Refractory period on interruption. Upon detection of an incoming call the oscillation period will be lengthened by 0.05 seconds.
- **s**, the phase response curve (PRC) slope factor. It determines how fast the animal recovers from inhibition, this is an experimental value between 0 and 1.

In the paper the model is implemented using a Monte Carlo [48] simulation to simulate the real-time aspect of signal delay based on inter-agent distances, the paper does not delve into the implementation details of the simulation.

The simulation enabled testing with different values for the PRC slope (s). In the paper experimentation was reported for values of s ranging from 0.1 to 1.

In effect, s is an expression of the model's ability to recover from inhibition. The lower the value, the faster the animal can recover. In theory a suitable value for s can help the animal gain precedence on the signals of other active males on the following oscillation by adjusting the period for the current one.

To evaluate the performance of the model, a set of preference factors was defined for an observing female that was placed in the simulated environment.

The female's main goal is to record whenever two or more calls are made after a call is detected (β) and before a certain time (γ) has gone by. For chorusing males the one with the earliest call within that period is considered the attractive male.

14.2 Translating the Model

To illustrate the viability of Rana as a tool for the kind of experimentation presented in the paper, the model has been translated into a corresponding

Rana agent. The model is translated to a function that correspond with the agent modelling paradigm of Rana. The first section of the model deals with call reception and is handled whenever a call is perceived by the agent during its handle-event phase, while the last section is handled internally during the agent's take step-phase.

14.2.1 Method

Implementing a purely mathematical model such as this is fairly straightforward in Rana. However there is some level of abstraction needed as the implementer gets some things for free, such as the real-time propagation of call events. The event makes it possible to define a propagation speed and distance naturally defined by agent placement in the environment. The Rana agent design paradigm also allows for reactive behaviour to be written into the model separate from the main oscillation as the signal is received outside regular agent take-step flow, that is during the handle-event phase.

The male acts as an oscillator that in every period will fire off a signal with duration x. For expository reasons the model is implemented in the following sections as three iterations of increasing complexity.

- **Free-running oscillator**, that has a variable period. This model will be used for experimentation.

- **Inhibiting oscillator**, that resets its oscillation upon reception of another agent's signal. It serves as a prelude to the final model.

- **Inhibiting oscillator**, that has a phase response value which adapts to incoming signals using the phase response curve value s, representing the complete model with behaviour corresponding to the one presented in the paper.

14.2.2 The Free-running Oscillator

The free-running model oscillates between 0 and 1 with a period of $T + \epsilon$, where $T = 500[ms]$ and ϵ is a stochastic variable with deviation of 0.30[s] and a standard mean of 0[s]. The value for ϵ is generated using Rana's statistics module.

To accommodate Rana's simulation design the oscillation is driven by the take-step phase. To support additional variables are introduced.

- S. Step precision of the simulation, 1[ms].
- T_n. Oscillation time of the current take-step phase.
- $\bar{T_n}$. Oscillation time of the previous take-step phase, used to move the oscillation time forward, so at every take-step $T_n = \bar{T_n} + S$ is evaluated.
- O_l. The current oscillation level, for presentation purposes it has a value between 0 and 1.

The model will emit a call event at $T_n = T_t$ with a propagation speed of $v = 343[m/s]$.

The pseudo code for the models take-step phase can be seen in listing 14.1. As the model's oscillation is not affected by outside calls it has no handle event phase.

To inspect the models performance it will collect data on its oscillation and write it to a data file. Figure 14.1 shows a data plot for a single oscillator.

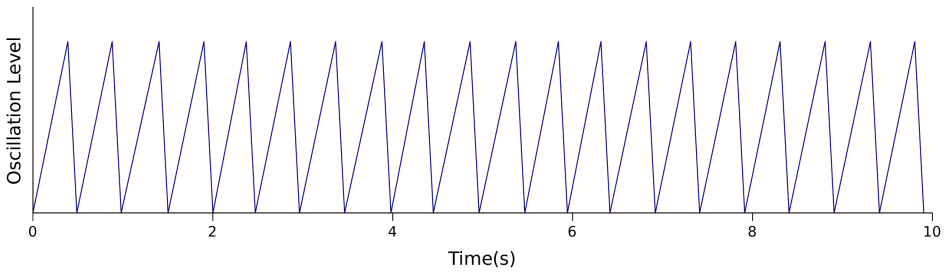


Figure 14.1: Oscillations for a time period of 10[s]. The model emits an event at peak which happens at $T_t - r$

```
add take step precision to T_n

if T_n > T_t then
    set Called to false
end

if T_n >= Tt - r and Called is false then
    emit event
    set Called to true
end
```

Listing 14.1: Pseudo code for the agents take-step phase. Notice how a Called boolean has been introduced to ensure that the agent only calls once per oscillation. As the the targeted oscillation time and step-precision of the simulation is defined as reals the agent cannot be sure that T_n can become equal to T_t, as it might overshoot. This model takes that into account by using the Calling boolean.

14.2.3 Inhibiting Oscillator

This model will exhibit inhibition on an incoming signal by resetting the oscillation with a small delay (y). The effect on an incoming call is that the oscillator will reset its oscillation timer and start over, which means that $T_n = 0$ and $T_t = T + \epsilon$.

Due to the real-time event propagation engine of Rana, section $[d + l/v - r - t]$ of the original model is equal to the arrival time of an external call. This means that when an agent emits a call, Rana will propagate that event to all other oscillators and initiate their event-handle function when the call has arrived at their position.

The Rana model of the inhibiting oscillator both has a take-step phase and a handle-event phase that implements the effect of the external call. Pseudo code for the take-step phase can be seen in listing 14.2. The handle-event function can be seen in listing 14.3.

```
add take step precision to T_n

if T_n >= T_t − r and Called is false then
    emit call event
    set Called to true
end

if T_n >= T_t then
    set T_t to T + ε
    set T_n to 0
    set Called to false
end
```

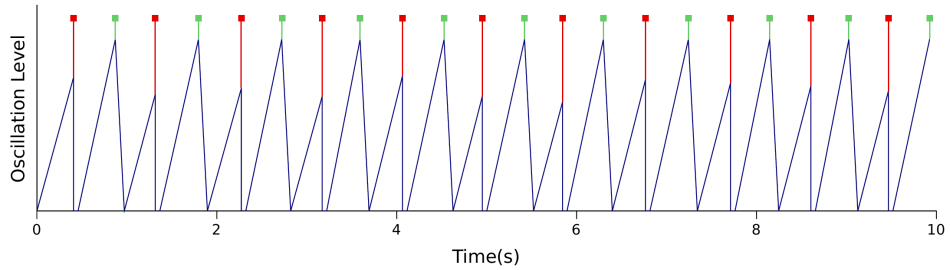
Listing 14.2: Pseudo code for the inhibiting oscillators take-step phase. For this one we have again the Called boolean, aside from its original function it now also ensures that the agent is only inhibited on the upward flank of the oscillation.

```
if Called is false then
    set T_t to T + ε + y
    set T_n to 0
end
```

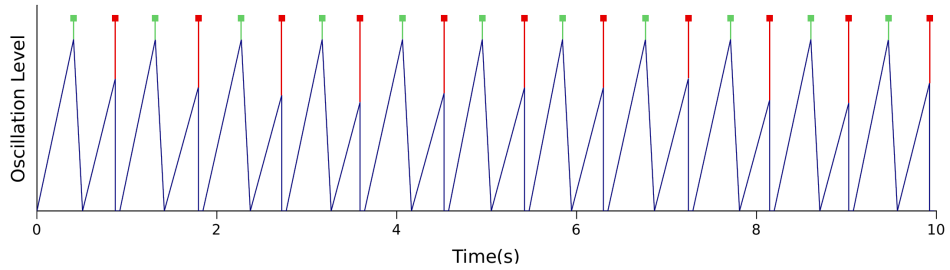
Listing 14.3: Pseudo code for the handle-event phase. On an incoming call the oscillation period will reset ($T_n = 0$) itself, and a new target oscillation time is calculated (T_t), with an added delay y.

Figure 14.2 shows the oscillations of two agents over a period of 10[s]. They are placed with an inter-agent distance of 10[m]. The oscillators emit a call at the peak of the oscillation.

During the simulation the two inhibiting oscillators will alternate between each other exhibiting antiphonal behaviour. Experiments with adding more agents results in a less orderly simulation, as one agent can be inhibited indefinitely by two or more agents with better precedence. Precedence for this model is determined by the initial calculation of the oscillation period of T_t.



(a) Oscillator 1



(b) Oscillator 2

Figure 14.2: Two oscillators, data is collected over a 10[s] time period. As they inhibit each other with their calls they will alternate thus creating an antiphonal chorusing mechanic. Each graph has two coloured lines. ■ marks the emission of a call. ■ marks detection of an inhibiting call.

14.2.4 Introducing the Phase Response Curve

To to fully implement the model and enable it to adapt its signal cycle to inhibiting signals for both increasing and decreasing oscillations two new elements are introduced. The PRC parameter and anticipatory action. It means that the agent is able to reset the oscillation on both the rising and downward flank of the oscillation. An oscillation reset cannot occur during a call which happens in the last part of the downward flank at $T_n = T_t - r + x$. This corresponds to realistic animal behaviour as most chorusing animals have severely reduced hearing during call emission.

The goal of the model is to gain precedence on fellow agent calls events which will serve increase its overall attractiveness to females in the near field of the chorus.

The PRC is an experimental value between 0 and 1. At 0 no delay is added

upon reception of an inhibiting signal. A high value for s will increase the males chance to take precedence and more closely match the call of a different male. A low value of s It can also potentially increase the individuals call frequency in a chorus when compared to the previous model. This is due to the oscillation period following an interruption will be shorter than the previous one.

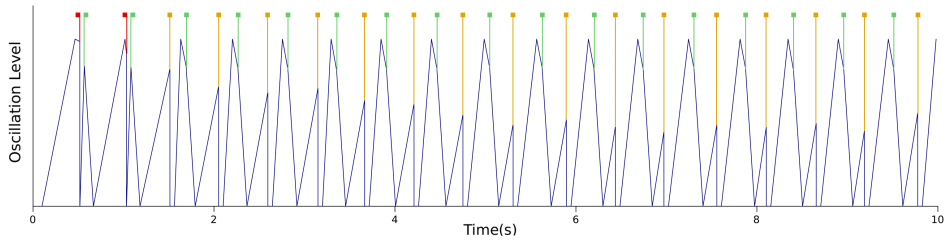
```
add take step precision to T_n

if T_n >= T_t - r and Peaked is false then
    emit call event
    set Peaked to true
end

if T_n >= T_t then
    set T_t to T + ε
    set T_n to 0
    set Peaked to false
end
```

Listing 14.4: Pseudo code for the PRC oscillators take-step phase..

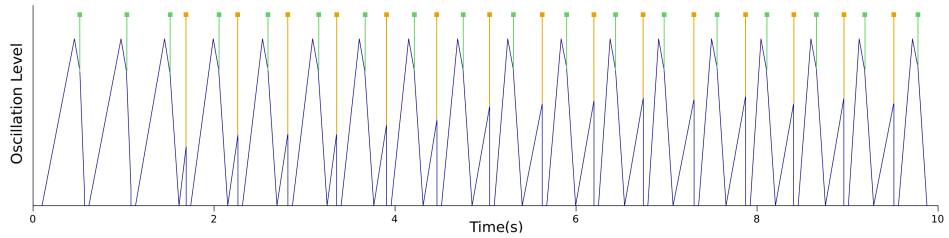
```
if Tn is false then

    set T_t to T_n × s + T_t
    set T_n to 0

    if oscillator has peaked then
        set T_n to T_t - y
    end
end
```

Listing 14.5: Pseudo code for the handle-event phase

The model exhibits a more dynamic period initially, than the previous one, depending on the value of s. Its behaviour is illustrated in figure 14.3. The figure displays two different individuals in the same simulation, again placed 10[m] apart. A natural adaptation to incoming signals occur after a short initial period of recovery and interruption. These two agents eventually settle into a rhythm, despite the first agent's recovery mechanism it will trail the second agent throughout. Experiments with lower values of s has shown that alternation becomes more dynamic.



(a) Oscillator 1



(b) Oscillator 2

Figure 14.3: Two inhibiting oscillators with PRC. Here the PRC is relaxed with a value of $s = 0.5$ which allows the model to recover from inhibition fast. Oscillator 1 had an initial longer running time, which resulted in inhibition from a call emitted by oscillator 2. The call was intercepted on its downward flank, before it started making a call.
Compared to oscillator 1 it is evident that oscillator 2 had the initiative with an initial smaller T_t.
■ Marks detection of an inhibiting call on the upward flank. ■ Marks detection of an inhibiting signal on the downward flank, it can only be interrupted in the first section, when it is not emitting a call. ■ Marks the emission of a call.

The complete Rana agent's Lua code for this model can be reviewed in the appendix in listing B.1 through B.3 on page 219.

14.3 Validation Experiments

To validate the translation of the model towards the work presented in the paper two experiments where performed. In the paper experiments where based on the total number of calls emitted, here we deviate from that and use a simulated runtime of 100[s] instead. The reason for this deviation is that it lends an extra way to evaluate a simulation, by counting the number of calls made with-in a fixed period.

The precision for the take-step phase is set to 1[ms] and the handle-event precision is set to 1[μs].

The experiments where the following:

- Test synchronization for different values of s using two male callers. This will test whether it is possible to reproduce the results presented in the paper.
- Test effect of inserting a free running agent into the preceding simulation. This will test how the model deals with inhibiting independent sources.

The experiments features three different agents.

The Male is an inhibiting oscillator adopting the model described in section 14.2.4. This is the main research target, and has designation **R**.

The Free-running Oscillator adopts behaviour of a free-running oscillator described in section 14.2.2, and has designation **I**.

The Female evaluates the simulation results, a simple data collection agent in the guise of a female has been implemented to evaluate call synchronization. It is placed in the environment, with equal distance to the active **R** agents. The female will flag calls as synchronized if they fall within a period between $\alpha = 5[ms]$ and $\beta = 30 + x[ms]$. α represents a short delay after the leading male.

14.3.1 Synchronization Experiment

Greenfield ran this experiment with two **R** agents placed 10[m] apart, and a female placed with equal distance to the two callers.

Synchronization between two calls is achieved if two successive calls propagate to the female within a period of 80[ms]. The female records the total

number of calls as well as number of calls received in synchronization with a preceding call.

14.3.1.1 Results

We have run experiments with 9 different values for the PRC slope (s) ranging from 0.1 to 1.0. Each experiment was run 5 times. Figure 14.4 displays a box-plot graph which illustrate the range of performance for different value of s. A new value for ϵ was generated on the start on every new oscillation period, it is an individual value for each agent.

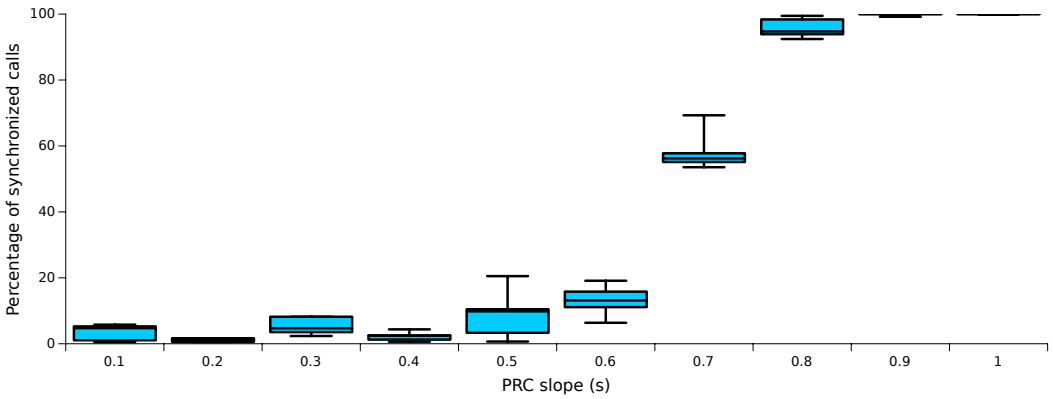


Figure 14.4: Graph with box-plots illustrating performance for different values of s, for a chorus with two active **R** agents. Each of the box-plots illustrates the percentage of calls received in synchronization with a fellow caller.

The total number of calls throughout all experiments ranged from 320 to 370.

In his paper Greenfield suggests that synchronization, or rather chorusing, will start to be evident for s values above 0.5, as this experiment supports that claim, as there is a clear increase in the percentage of synchronous calls starting from s=0.6. Near perfect chorusing is achieved for $s >= 0.8$.

At lower values for s the agents exhibit antiphonal behaviour (the calls alternate) and synchronization seems to be more sporadic.

14.3.2 Introducing the Free-running Oscillator (I)

So what happens if we introduce agent **I** to the environment as an invading species. This agent has the same oscillation period as agent **R** however it has no regards for fellow callers and thus it can not be inhibited.

A singe **I** agent is placed with equal distance to the two males, forming a triangle with the female in the centre.

14.3.2.1 Results

Using the same data collection parameters from the previous experiments a box-plot graph has been generated displaying the number of synchronized calls versus the total number of calls emitted, see figure 14.5.

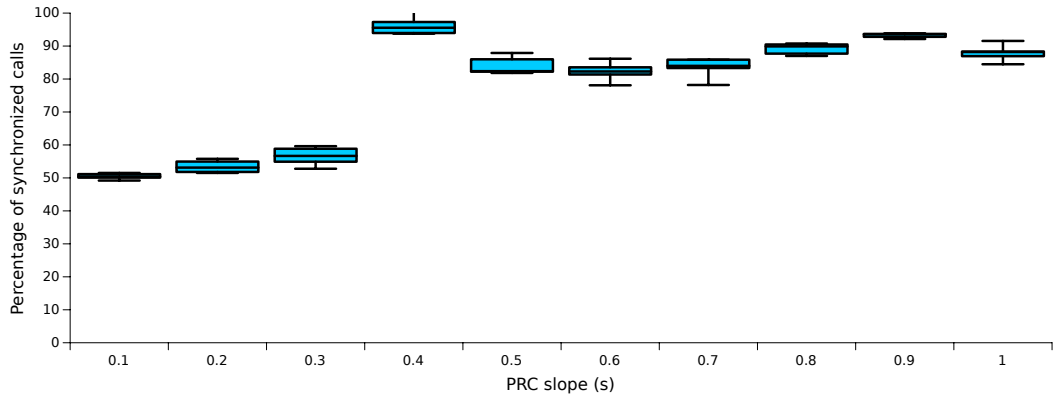


Figure 14.5: Graph with box-plots illustrating performance for different values of s, for an experiment with two **R** agent and a single I agent. Each box-plot illustrates the percentage of calls received in synchrony with another call

Something interesting happens with the performance of agent **R** as the value of s increases. For s= 0.4 and below the number of calls each **R** agent emits is relatively equal and remains at an average of 150 for each. Starting from 0.6 one of the **R** agents remains uninhibited and still makes 150 calls on average, however the other male will be inhibited only emitting very few calls. As expected agent **I** has a steady average at 200 calls throughout the experiments.

This means that the box plots of figure 14.5 do not tell the whole story. To further illustrate the phenomena of the inhibited male a plot presenting the total amount of calls can be seen in figure 14.6.

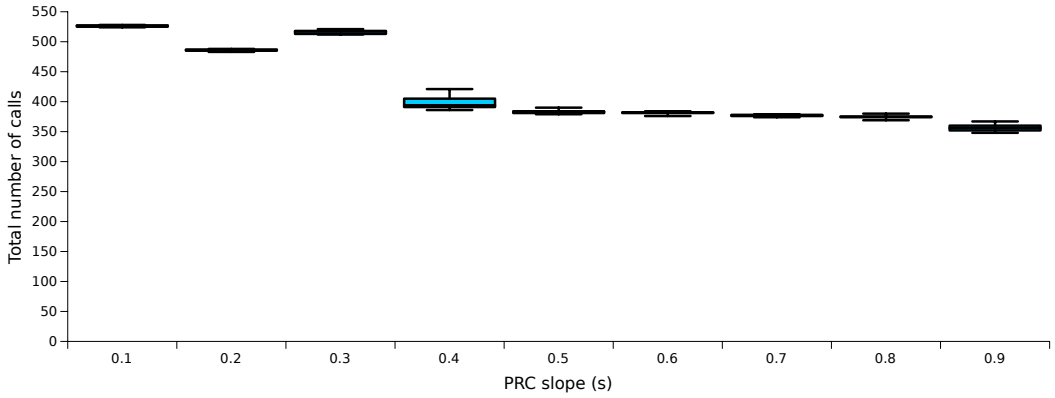


Figure 14.6: Box-plots, one for each value of s, for an experiment with two **R** agents and a single I agent. Each box-plot illustrates the total number of call emitted.

While synchronization is more sporadic at lower values of s the total number of calls is high as both **R** agents have a good chance of recovering from inhibition. At higher values of s one agent is nearly constantly jammed and the number of total calls drops significantly. The upside is that the **R** agent that is not inhibited manages to achieve good synchronization with the free-running agent.

These results support the ones achieved via the Monte-Carlo simulation in the paper [25]. It states that agent **I** is an inhibitor that can achieve a higher number of calls than its peers while managing to completely jam one of **R** agents when they have low values of s.

Further experiments have been performed by introducing ever increasing numbers of free running oscillators into the population. The end result is that both **R** agents can only emit sporadic calls but are otherwise completely inhibited regardless of the PRC value. This is true for **I** agent numbers greater than 3.

14.3.3 Conclusion

With the two experiments we have managed to support the original papers results by translating the model to the Rana agent modelling paradigm. During experimentation there was at no point any noteworthy discrepancies between the Rana model and the results listed in the paper.

14.4 Discussion

While the Monte-Carlo simulation used in the original work proved to be a suitable tool for validating the model, the concept is constrained as it is developed specifically for that model supporting a limited number of scenarios.

Adapting the model to be used in a real-time MAS simulation tool makes it possible to offer continued development and experimentation. Now that the Rana model has been validated with respect to the original, it is now an option to translate the Rana model into an agent module for use in composite models. For example, it can be used as the calling state for the foraging model we presented in the Rana demonstration chapter, on page 91. This can help increase the usefulness of the model, because in nature singing animals do more than just chorus, they forage and rest, something that could be implemented into the agent's behaviour as future work.

The consequence of adapting models to the Rana paradigm is that it is now possible to breathe new life into other published biological simulations in order to make the modelling and further experimentation more accessible. One case could be as the simulation of *Puerto Rican Treefrog* chorusing from 1989 using computing networks [8]. Another, perhaps more accessible, one is a proposed neural network frog model [58] for sexual selection in the female túngara frogs, which could be used to expand the simple female presented in the Greenfield paper.

Enabling chorus simulations has been a development constraint for Rana since its inception. In this dissertation Rana's design and implementation has

revolved around the male frog calling behaviour. However, by implementing an insect calling model such as this marks a very significant milestone for Rana as a general tool for performing real-time critical simulations. This model can also be relevant for simulating singing frog species by adjusting the various parameters.

14.5 Conclusion

Using Rana's real-time simulation paradigm, it has been possible to transform a proposed mathematical model on chorusing and provide proof that a Rana agent can be designed to exhibit the same behaviour as the one presented in the paper. This makes it possible to explore more acoustic models and provide a common denominator for working with these models that uses Rana for simulation and validation for the models.

IV

Conclusion

"They were patient, but they were not yet immortal. So much remained to do in this universe of a hundred billion suns, and other worlds were calling. So they set out once more into the abyss, knowing they would never come this way again"

2010, Arthur C. Clarke

Chapter 15

Requirements Evaluation

Here we will evaluate Rana and its expansion against the initial requirements defined in the introduction. Initially two artefacts that comprise the MAS simulation were introduced, the modelling paradigm and the simulation interface.

15.1 The Modelling Paradigm

To satisfy the need for a sensible modelling paradigm Rana has introduced two major design artefacts: the agent and the event.

- **The Agent**. In Rana the agent is a representation of an independent entity, that can interact with other agents and the environment. Rana supports two types of agents.
  - **Active**: These agents represent objects of study. For example, in the chorus simulation, the calling males and listening females are active agents, in the traffic simulation drivers and traffic-lights comprise the active agents.
  - **Utility**: Examples of this type include: data collectors that are designed to collect simulation data by requesting data points active

agents or by intercepting agent events, environment controllers that constantly seed food items to the environment for foraging agents, and master agents that sets up the simulation. These three types of agents all fall in the utility category, because they have no active role in the simulation.

- **The Event** is a construct that allows for agents to communicate and generally express external actions. Events are part of the real-time aspect of Rana, as they can be made to propagate with variable propagation speed denoted in metres per. second.

  Agents can also define event-processing functions that specify the nature of an event's propagation. For example, this can be used to define directional sound emissions such as a bat's call.

  Events are highly flexible as they can hold data tables, so they can also be used for agent configuration and data collection by utility agents.

15.2 Simulation Interface

Rana's simulation interface offers a graphic user interface that allows for configuration of both the simulation and the environment.

- **Configuration**. A simulation can be configured via Rana's configuration panel. This allows for control over simulation run-time and precision levels in relation to simulation of physical time, as well as other simulation parameters.

- **Visualization**. Three modes of simulation output are supported.

  - Text output; allowing the agents to transmit text messages to the user interface.

  - Live simulation observation; via the live view panel, where the environment is represented as an bitmap image and the agents each

have an avatar rendered as a small coloured Xs' with the option to display the agents unique runtime identity codes along with their avatar.

– Event intensity and activity; via the event visualization functionality.

15.3 Tool Requirements

Expanding the general requirements for MAS simulation, a number of tool specific requirements were also defined.

1. **Simulation of physical time**. Rana's simulation core ensures that agent actions and interactions are synchronized. To enable this Rana operates with two phases.

   - **Handle-event**. Reaction to events emitted by other agents. Offers very high resolution with minimal impact on simulation performance, the default precision level in Rana is a microsecond.
   - **Take-step**. This has more coarse granularity, with a default precision level of a millisecond. The take-step phase allows agents to take delayed actions in response to internal decisions. It is also in this phase the agent moves and take internal actions.

2. **Agent modelling**. The Rana agent design paradigm offers a flexible and powerful modelling language via the Lua dynamic programming language. Agent design is thus separate from the compiled simulation core. This means that agents can be designed and modified during the Rana runtime.

   Furthermore, it is also possible to augment agent design by developing and expanding agent modules in Lua in runtime.

Agent modelling is further supported by a central API that offers a wide variety of functions, for example, central data storage, environment surveying, agent manipulation and collision detection.

3. **Benchmarking**. In Rana, observation of emergent properties within the simulation is enabled via its visualization options and data collection agents, that can generate simulation specific data for processing in third party programs.

Chapter 16

Discussion

Throughout this dissertation there have been many discussions, ranging from implementation details to Rana's place in the current state of the art. this chapter will attempt to summarize these discussions and bring to light the most important points. This chapter will cover each separate part in sections and then wrap up the discussion in a final section.

16.1 Rana

16.1.1 Parallel Discrete Event Switching

Rana is designed towards simulation of physical time by representing the flow of time through series of snapshots, each representing either a take-step or the handle-event phase. For example, this snapshot mechanic means that in Rana's multi-threaded simulation, two agents with collision detection can attempt to move to the same position in the same take-step phase breaking the collision mechanic. This is a known problem with parallel discrete event-base simulations [22].

More specifically in Rana this phenomena can arise from three different

agent actions, collision detection, map manipulation and event handling. For collision detection and map manipulation Rana offers a couple of options to deal with this problem.

16.1.1.1 API synchronization

The API has a *checkandmove* function that ensures that agents only move if the position is free and a *checkandchange* function for modifying the environment only if it has a given value.

While these functions can ensure good synchronization and prevent undefined behaviour, they use a mutex that locks out any change or checks on the environment until the function has returned. Heavy use of these two functions throughout development can then basically break the performance advantage gained from multi-threading the simulation.

For simulations with only sporadic environment manipulation and a more relaxed collision detection paradigm these two functions provide a very easy to use interface for avoiding agent-action ordering ambiguity.

16.1.1.2 Utility Agents

As Rana has support for utility agents, a sort of meta agent can be set up to handle movement and environment manipulation. For example, agents can request an action to change a section of the environment by sending an event to an environment control agent that handles all incoming requests and decide on which action to take.

It could even be possible to split the environment into sections, where multiple meta agents control a different section of the map.

This solution will take more effort to implement. However it offers a good deal more flexibility than the API functions and can be implemented to complement Rana's multi-threaded simulation. In particular, it offers the user freedom to choose explicit policies for handling ambiguous situations (typically race-conditions).

This solution can also be applied help solve ambiguity on event-handling (if more than one event arrives in the same phase).

16.1.2 Event Propagation and Movement

Rana's event propagation engine processes each event's arrival time and marks it by generating a reference, for each receiving agent, that holds the activation time at the receiving agent's position.

This can pose a problem for simulations with moving agents that either move fast or are separated by long distances, as the event propagation times registered by the Rana core are no longer relevant.

16.1.2.1 Agent Based solution

It is a limitation that can be worked around with some creative agent design. By emitting events with a propagation speed of 0, the event will be triggered on the next handle-event phase. The agent can then store the event information and choose when to act on the event data.

It is a goal to offer a module for handling this problem using the above mentioned solution. However, it is important prior to designing and implementing a simulation to consider whether this would pose a thread to the integrity of a simulations output.

16.1.2.2 Time Reversibility

In more general terms, this is the same problem encountered when simulating robotics, where one cannot predict when contact will occur and the kinematics will change [77]. The solution used for robotics is based on time reversal [30], basically rolling back time with the knowledge in hand to fix the problem on the next pass.

However a multi-agent simulation can feature several hundred agents so implementation of time-reversal can represent a performance problem.

16.1.2.3 Simulation Core Event Tracking

A better solution would be to allow the simulation core to track moving agents and update event arrival times if the agent is moving. But like time-reversal, this is a significant implementation task.

16.2 Event Processing and Visualization

We have introduced two new elements for expansion of the Rana tool, the event-processor for a unified interface for agent event evaluation and the event-map that enables event visualization by using the functionality defined by the event-processor.

16.2.1 The Event Processor

The advantage of having an interface for determining event intensity is immediately apparent. As illustrated in the event processing demonstration chapter, an agent can use an implementation of the event processor to determine neighbour caller relevance, which can serve to enable more realistic scenarios.

For example, the Greenfield model does not scale well with increased numbers of invasive free running oscillators, however in nature there are examples of functioning multi-species chorusing [51]. The event-processor can be used for filtering of incoming events in correspondence with published data on multi-species chorusing.

And while it can be argued that the event-processing function is limited with only two pre-defined return values, it is possible to implement more return values, without compromising its functionality in regards to event visualization, thanks to the flexibility of the Lua programming language.

16.2.2 Event Visualization

The concept of visualizing events as coloured intensities that are processed via an event-processing implementation can seen as rather abstract. However, it can provide a unique view into the event scape and the event driven mechanics of the simulation, it can also be used to evaluate whether an event-processing implementation processes event propagation correctly.

16.3 Evaluation

Rana has been evaluated both against state of the art and as a tool for simulating three very different fields.

16.3.1 State of the Art

Rana is developed with a very different mindset from the current state of the art. It has been developed with a focus on the constraints posed by simulation of a acoustically driven biological system. Never-the-less, the Rana modelling paradigm and simulation tool has proven to be flexible enabling it to compete with the state of the art.

This claim is supported by the three following modelling chapters, each of which demonstrates Rana as fully capable MAS simulation tool capable of providing suitable modelling and simulation, along with a flexible tool-set for benchmarking these simulations, via its visualization and utility agents.

Chapter 17

Future Work

While Rana represents a complete tool that enables simulation of both general and real-time MAS simulation, there are a number of both potential and an ongoing projects that can provide fruitful expansions to Rana.

17.1 3D and Physics Support

As part of a M.Sc. thesis project, Rana is currently being expanded to enable physics simulation. The project's goal is to perform experiments with multi-agent dynamics of artificial satellites orbiting celestial objects such as the asteroids and the ISS space station.

To support the physics, the attributes shared between the agent and the simulation core have been expanded to include, position in the z dimension, mass, charge and radius. Furthermore, Rana's events and event handling mechanics have been updated to perform correct event propagation calculations in the third dimension.

Rana's live view has been updated to support 3D rendering of agents. The new 3D view has been integrated into the existing view panel, replacing the 2D view as needed. The 3D visualization is rendered and implemented in via

the OpenGL [64] framework.

17.1.1 Physics Integration

To enable simulation of physics a central physics engine is implemented. The physics engine integrates into Rana's core, and can operate at the granularity of the movement precision of the agents (which is equal to the take-step precision).

Basically, when an agent activates movement towards some destination a path and its movements effect will be processed by the physics engine, which will then control the positions of all agents affected, including the original one, with some degree of granularity. This enables simulation of small objects orbiting a large object for the orbital mechanics simulation and will even support simulation of larger celestial objects.

Integrating the physics of orbital mechanics is done by the introduction of a new API call, where an agent can opt in on being part of the global physics engine. The physics engine is enabled if two or more agent opt to be part of the physical world.

In that case the physics engine is automatically called on every take-step phase, which matches the agent movement and internal action precision level.

17.2 Environment Expansion

As discussed, the Rana map implementation is very limited in function. It is a goal to offer a much more comprehensive environment interface, that can support all the variables needed to fully realize a natural environment.

17.2.1 Weather

Weather is a significant factor to consider when simulating biological systems, for example, anuran reproduction can be affected by various weather effects

such as temperature, rainfalls and wind [55].

Expanding Rana with an API for generation of weather patterns, potentially via a stochastic model [85] controlled by a weather agent, poses an exiting yet to be explored avenue for expanding its functionality towards simulation of biological systems.

17.2.2 Map Support

As discussed in the chapter on traffic light simulation, integrating support for OpenStreetMap data files (OSM), will offer a way to represent real environments, even non-city specific ones. Furthermore, OpenStreetMap will present itself nicely as visual representation for a live simulation view.

17.2.3 The 3D Map

Solution towards 3D map representations for robot systems exists, one of these is the OctoMap [86], a probabilistic environment representation, that can be used for feature extraction and visual representation in a live-view.

Implementation of such an environment representation combined the the aforementioned 3D expansion for Rana would further strengthen Rana's simulation capabilities for 3D enabled simulations.

17.3 Simulation Configurability

The current Rana interface is, aside from a few short-cut keys, mouse driven. It is currently not possible configure successive simulations. This is a problem as constant user interaction is a nuisance (to the user) and because mouse driven user interfaces are known to cause acute pain [1].

It is therefore a goal to enable Lua configuration of Rana, so instead of loading a master agent the user loads a configuration file that defines: precision levels, run-time, master-agent, map options and a number of successive

simulation runs. This could potentially ease user-interaction workload and allow the user to focus on other more important tasks.

17.4 Bridging Multi-channel Recordings and Modelling

Work has been published towards separation of callers using microphone arrays [35], and research has been performed on localization of individual chorusing green treefrogs (*Hyla Cinerea*) [34]. This brings a new avenue towards development and validation of proposed models against real references.

At University of Southern Denmark a battery-powered open-source multi-channel recording array has been developed, based on the research of Thor Andreassen [2]. The system is build around a high performance digital analogue converter board (DAC) [60] and the Rasberry Pi micro-PC [20]. The system can, in its current iteration, record at 3[Mhz] shared across 8 channels.

This system has been successfully deployed and tested on the roofs of the University of Southern Denmark where it is doing long-term bat recordings. It has also been successfully used to record frog chorusing of the gray treefrog (*Hyla Versicolor*) in Minnesota, and the green treefrog (*Hyla Cinera*) in Texas, in the summer of 2015.

In relation to Rana the longterm goal is to use the methods employed by Douglas L. Jones and Rama Ratnam [34] [35] to localize individual sound emitting animals, towards development and validation of models using Rana as the modelling and simulation tool.

A successful interfacing of Rana models and real recordings will help bring the simulated MAS and the biological system closer together.

17.5 Graphic Agent Design

Even though much effort have been done towards making implementation of agent behaviour in Rana accessible by using a modern dynamic programming language, and providing a modular design approach where agent states are separate modules. Agent design remains a complex task and more should be done to make Rana agent modelling more accessible and user friendly.

A good path to take towards this, would be to use the Rana modular design approach. Which can be utilized via a graphic domain specific language and a design interface which enable users to define active agent states and their transition rules. The interface could also embed editing of agent module code, for more advanced use-cases.

This could possibly be enabled by adopting the Logo [57] programming paradigm (as Repast has done with Relogo).

Chapter 18

Achievements

The work presented in this dissertation represents the following achievements.

- Successfully implemented a high-performance real-time open-source simulation platform named Rana. Rana that encompass the following elements.
  - A flexible agent design paradigm, using a dynamic implementation language, with equal focus on agent and interaction design.
  - A real-time engine, that enables simulation of multi-agent systems with real-time constraints in both agent action and propagation of events.
  - Visualization of both the simulation and agent events.
- Successfully deployed Rana as the course-ware for a multi-agent systems computer science course at University of Southern Denmark in 2014 and 2016.
- Successfully used Rana in varied scientific simulations; ranging from an acoustic driven chorusing modelling to autonomous mining robots.

IV. Conclusion

- Scientific recognition with two published papers (both are included with this dissertation from page 248):
  - On Rana the agent design paradigm in relation to frog chorusing [37] in the conference preceding of SAB 2014 (Simulation of Adaptive Behaviour).
  - On Rana's simulation approach [38] in the conference preceding of IAT 2015 (Intelligent Agent Technology).

Chapter 19

Summary

Throughout this dissertation we have described the development and utilization of a new MAS simulation tool.

The work consisted of three major parts:

- **Rana**. Describes the design, implementation and demonstration of a MAS simulation tool that encompass the following.
  - An agent modelling paradigm with focus on agents and events (events is a term for perceivable agent actions, such as a frogs call).
  - Multi-threaded simulation core that supports real-time agent actions and propagation of events.
  - Introduction of a utility agent modelling concept that enables design of meta-agents which can be used for tasks such as benchmarking and simulation set up.
  - Graphic user interface, that allows for environment definition and simulation visualization.
- **Event-processing**. Describes the design, implementation and demonstration of an event-processing paradigm that provides a common interface for designing behaviour-based event interpretation.

Event processing is also a means to enable event visualization via an expansion of the Rana tool.

- **Evaluation**. Rana was evaluated against the current state of the art, and three different Rana models where described.

19.1 Rana Availability

As a final conclusion, the complete Rana tool is published online using the github platform. The Rana repository consists of the following elements.

- Releases, as Rana is being developed, release snapshots are released and pre-compiled packages are provided for the windows[1] platform.
  (link: https://github.com/sojoe02/RANA/releases).
- Documentation, Rana is documented using the github wiki, here you can find guides for modelling, installation and the user interface.
  (link: https://github.com/sojoe02/RANA/wiki).
- Source code, Rana's source-code is freely available and distributed via github.
  (link: https://github.com/sojoe02/RANA).

Rana currently provides 12 different demonstration models, as part of its source code and releases, they are listed in appendix on page 223.

[1] It is quite easy to compile Rana on Linux platforms, the Rana wiki provides an easy-to-follow guide for it.

Bibliography

[1] J. H. Andersen, M. Harhoff, S. Grimstrup, I. Vilstrup, C. F. Lassen, L. P. Brandt, A. I. Kryger, E. Overgaard, K. D. Hansen, and S. Mikkelsen. Computer mouse use predicts acute pain but not prolonged or chronic pain in the neck and shoulder. *Occupational and environmental medicine*, 65(2):126–131, 2008.

[2] T. Andreassen, A. Surlykke, J. Hallam, and D. Brandt. Ultrasonic recording system without intrinsic limits. *The Journal of the Acoustical Society of America*, 133(6):4008–4018, 2013.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[4] B. Arons. A review of the cocktail party effect. *Journal of the American Voice I/O Society*, 12(7):35–50, 1992.

[5] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[6] P. J. Bishop, M. D. Jennions, and N. I. Passmore. Chorus size and call intensity: female choice in the painted reed frog, hyperolius marmoratus. *Behaviour*, 132(9):721–731, 1995.

[7] J. Blaxter, J. Gray, and E. Denton. Sound and startle responses in herring shoals. *Journal of the Marine Biological Association of the United Kingdom*, 61(04):851–869, 1981.

[8] J. S. Brush and P. M. Narins. Chorus dynamics of a neotropical amphibian assemblage: comparison of computer simulation and natural behaviour. *Animal Behaviour*, 37:33–44, 1989.

[9] J. Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[10] D. J. Cook, M. Youngblood, and S. K. Das. A multi-agent approach to controlling a smart environment. In *Designing smart homes*, pages 165–182. Springer, 2006.

[11] G. Cordasco, A. Mancuso, F. Milone, and C. Spagnuolo. Communication strategies in distributed agent-based simulations: The experience with d-mason. In *Euro-Par Workshops'13*, pages 533–543, 2013.

[12] J. L. T. Da Silva and Y. Demazeau. Vowels co-ordination model. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 3*, pages 1129–1136. ACM, 2002.

[13] R. Darwin Charles. On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life. *Murray, London*, 1859.

[14] S. E. Deering. Internet protocol, version 6 (ipv6) specification. 1998.

[15] Y. Demazeau and S. V. Jorgensen. Multi-agent system. summer course, 2014.

[16] J.-L. Dessalles and D. Phan. Emergence in multi-agent systems: cognitive hierarchy, detection, and complexity reduction part i: methodological issues. In *Artificial Economics*, pages 147–159. Springer, 2006.

[17] B. Falk, L. Jakobsen, A. Surlykke, and C. F. Moss. Bats coordinate sonar and flight behavior as they forage in open and cluttered environments. *Journal of Experimental Biology*, 217(24):4356–4364, 2014.

[18] A. S. Feng and J. Schul. Sound processing in real-world environments. In *Hearing and sound communication in amphibians*, pages 323–350. Springer, 2007.

[19] J. Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[20] R. P. Foundation. Raspberry pi 3. 2012.

[21] T. E. Foundation. Eclipse, 2016. [Online; accessed 1-Jan-2014].

[22] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[23] H. C. Gerhardt. Female mate choice in treefrogs: static and dynamic acoustic criteria. *Animal Behaviour*, 42(4):615–635, 1991.

[24] H. C. Gerhardt. The evolution of vocalization in frogs and toads. *Annual Review of Ecology and Systematics*, pages 293–324, 1994.

[25] M. D. Greenfield, M. K. Tourtellot, and W. A. Snedden. Precedence effects and the evolution of chorusing. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 264(1386):1355–1361, 1997.

[26] O. Gutknecht and J. Ferber. *The MadKit Agent Platform Architecture*, pages 48–55. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[27] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[28] J. C. Hall and A. S. Feng. Influence of envelope rise time on neural responses in the auditory system of anurans. *Hearing research*, 36(2-3):261–276, 1988.

[29] F. H. Hatano, C. F. Rocha, and M. Van Sluys. Environmental factors affecting calling activity of a tropical diurnal frog (hylodes phyllodes: Leptodactylidae). *Journal of Herpetology*, 36(2):314–318, 2002.

[30] W. G. Hoover and C. G. Hoover. *Time reversibility, computer simulation, algorithms, chaos*, volume 13. World Scientific, 2012.

[31] R. Ierusalimschy. Lua 5.2 reference manual, 2011–2013. [Online; accessed 1-Feb-2014].

[32] I. Jacobson, G. Booch, J. Rumbaugh, J. Rumbaugh, and G. Booch. *The unified software development process*, volume 1. Addison-wesley Reading, 1999.

[33] N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1(1):7–38, 1998.

[34] D. L. Jones, R. L. Jones, and R. Ratnam. Calling dynamics and call synchronization in a local group of unison bout callers. *Journal of Comparative Physiology A*, 200(1):93–107, 2014.

[35] D. L. Jones and R. Ratnam. Blind location and separation of callers in a natural chorus using a microphone array. *The Journal of the Acoustical Society of America*, 126(2):895–910, 2009.

[36] S. V. Jørgensen. Distributed agent modelling of frog chorusing behaviour, 2013.

[37] S. V. Jørgensen, Y. Demazeau, J. Christensen-Dalsgaard, and J. Hallam. Biomimetic agent based modelling using male frog calling behaviour as a case study. In *From Animals to Animats 13*, pages 88–97. Springer, 2014.

[38] S. V. Jørgensen, Y. Demazeau, and J. Hallam. Rana, a real-time multi-agent system simulator. In *2015 IEEE/WIC/ACM International Con-*

ference on Web Intelligence and Intelligent Agent Technology (WI-IAT), volume 2, pages 92–95. IEEE, 2015.

[39] A. Kesting, M. Treiber, and D. Helbing. Enhanced intelligent driver model to access the impact of driving strategies on traffic capacity. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 368(1928):4585–4605, 2010.

[40] S. Khan, R. Makkena, F. McGeary, K. Decker, W. Gillis, and C. Schmidt. A multi-agent system for the quantitative simulation of biological networks. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 385–392. ACM, 2003.

[41] G. M. Klump and H. C. Gerhardt. Mechanisms and function of call-timing in male-male interactions in frogs. In *Playback and studies of animal communication*, pages 153–174. Springer, 1992.

[42] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in action*, volume 1. Manning, 2007.

[43] D. M. Kotz. The financial and economic crisis of 2008: A systemic crisis of neoliberal capitalism. *Review of Radical Political Economics*, 2009.

[44] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan. Mason: A new multi-agent simulation toolkit. In *Proceedings of the 2004 SwarmFest Workshop*, volume 8, 2004.

[45] B. Lutz. Trends towards non-aquatic and direct development in frogs. *Copeia*, 1947(4):242–252, 1947.

[46] J. MacFarlane. Pandoc: a universal document converter, 2013.

[47] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys (CSUR)*, 18(1):39–65, 1986.

[48] C. Z. Mooney. *Monte carlo simulation*, volume 116. Sage Publications, 1997.

[49] D. R. Musser, G. J. Derge, and A. Saini. *STL tutorial and reference guide: C++ programming with the standard template library.* Addison-Wesley Professional, 2009.

[50] P. Narins, A. S. Feng, and R. R. Fay. *Hearing and sound communication in amphibians*, volume 28. Springer Science & Business Media, 2006.

[51] V. Nityananda and M. A. Bee. Finding your mate at a cocktail party: frequency separation promotes auditory stream segregation of concurrent voices in multi-species frog choruses. *PLoS One*, 6(6):e21191, 2011.

[52] M. J. North, N. T. Collier, and J. R. Vos. Experiences creating three implementations of the repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 16(1):1–25, 2006.

[53] M. J. North, C. T. Nicholson, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. Complex adaptive systems modeling with Repast Simphony, 2013. [Online; accessed 3-Feb-2014].

[54] M. J. North, C. T. Nicholson, and J. R. Vos. Experiences creating three implementations of the repast agent modeling toolkit, 2006.

[55] K. L. Oseen and R. J. Wassersug. Environmental factors influencing calling in sympatric anurans. *Oecologia*, 133(4):616–625, 2002.

[56] M. Pall. The luajit project. 2008.

[57] R. D. Pea. Logo programming and problem solving.[technical report no. 12.]. 1983.

[58] S. M. Phelps and M. J. Ryan. Neural networks predict response biases of female túngara frogs. *Proceedings. Biological sciences / The Royal Society*, 265(1393):279–285, Feb. 1998.

[59] I. Pollack and J. M. Pickett. Cocktail party effect. *The Journal of the Acoustical Society of America*, 29(11):1262–1262, 1957.

[60] B. Porr. Usbdux-fast, the only usb based daq with an open source linux driver available today. Open-source, 2016.

[61] H. Quastler. *The emergence of biological organization*. Yale University Press, 1964.

[62] M. S. Reichert and H. C. Gerhardt. Gray tree frogs, hyla versicolor, give lower-frequency aggressive calls in more escalated contests. *Behavioral Ecology and Sociobiology*, 67(5):795–804, 2013.

[63] G. D. M. Serugendo, M.-P. Irit, and A. Karageorgos. Self-organisation and emergence in mas: An overview. *Informatica*, 30(1), 2006.

[64] SGI. Opengl sdk. Open source license, June 2016. [Online; accessed 21-Nov-2016].

[65] V. J. Soeren. Kasterborous, agent based real-time broadcasting simulation framework, 2013. [Online; accessed 21-May-2013].

[66] M. L. Sørensen. Traffic light simulation using rana. http://www.gnu.org/licenses/gpl-2.0.html, 2015.

[67] M. V. Srinivasan, S. Zhang, M. Altwein, and J. Tautz. Honeybee navigation: Nature and calibration of the" odometer". *Science*, 287(5454):851–853, 2000.

[68] J. Strang. *Programming with curses*. " O'Reilly Media, Inc.", 1986.

[69] B. Stroustrup. *The C++ programming language*. Pearson Education, 1995.

[70] K. Summers, R. Symula, M. Clough, and T. Cronin. Visual mate choice in poison frogs. *Proceedings of the Royal Society of London B: Biological Sciences*, 266(1434):2141–2145, 1999.

[71] A. Surlykke, S. B. Pedersen, and L. Jakobsen. Echolocating bats emit a highly directional sonar sound beam in the field. *Proceedings of the Royal Society of London B: Biological Sciences*, 276(1658):853–860, 2009.

[72] The-GNOME-Project. Gnumeric.

[73] The-Qt-Company. Qt project, 2013. [Online; accessed 21-May-2013].

[74] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[75] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[76] S. Tisue and U. Wilensky. Netlogo: A simple environment for modeling complexity. In *in International Conference on Complex Systems*, pages 16–21, 2004.

[77] D. Tolani, A. Goswami, and N. I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical models*, 62(5):353–388, 2000.

[78] M. D. Tuttle and M. J. Ryan. The role of synchronized calling, ambient light, and ambient noise, in anti-bat-predator behavior of a treefrog. *Behavioral Ecology and Sociobiology*, 11(2):125–131, 1982.

[79] J. Vinet and A. Griffing. Arch linux, a lightweight and flexible linux distribution. 2016. [Online; accessed 24-Nov-2016].

[80] A. Walker and M. Wooldridge. Understanding the emergence of conventions in multi-agent systems. In *ICMAS*, volume 95, pages 384–389, 1995.

[81] L. Wall, T. Christiansen, and J. Orwant. *Programming perl*. " O'Reilly Media, Inc.", 2000.

[82] H. Wallach, E. B. Newman, and M. R. Rosenzweig. A precedence effect in sound localization. *The Journal of the Acoustical Society of America*, 21(4):468–468, 1949.

[83] K. D. Wells. The courtship of frogs. In *The reproductive biology of amphibians*, pages 233–262. Springer, 1977.

[84] D. J. Wilkinson. Stochastic modelling for quantitative description of heterogeneous biological systems. *Nature Reviews Genetics*, 10(2):122–133, 2009.

[85] D. S. Wilks and R. L. Wilby. The weather generation game: a review of stochastic weather models. *Progress in Physical Geography*, 23(3):329–357, 1999.

[86] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard. Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, volume 2, 2010.

V

Appendix

"I have tasks enough for this day," said Death in a voice as heavy as neutronium. *"I can be robbed but never denied, I told myself, why worry?"*

Terry Pratchett, The Colour of Magic

Appendix A

Demonstration Appendix

A.1 Modules

A.1.1 Optimizing API calls via a module

```
local ranaLibAgent = {}

local currentColor = {r=255,g=255,b=255,alpha=255}
-- changes the color of an agent, returns true if the color
-- values are valid.
function ranaLibAgent.changeColor(options)

    local r = options.r or 0
    local g = options.g or 0
    local b = options.b or 0
    local alpha = options.alpha or 255
    local id = options.id or ID

    if r ~= currentColor.r or g ~= currentColor.g or b ~= currentColor.b or
        alpha ~=
        currentColor.alpha then
        currentColor.r = r
        currentColor.g = g
        currentColor.b = b
        currentColor.alpha = alpha
```

```
        l_changeAgentColor(id,r,g,b,alpha)
    end

end

return ranaLibAgent
```

Listing A.1: Excerpt from Rana's agent module, an API call will only be done if the new color is different from the previously requested one. Changing agent color is relatively slow as it requires user interface interaction.

A.1.2 API Error checking via modules

```
local ranaLibMap = {}

function RanaLibMap.radialMapScan(radius)

    local table
    if type(radius) == "number" and radius > 0 then
        table = l_radialMapScan(radius, PositionX, PositionY)
    end

    return table
end


return ranaLibMap
```

Listing A.2: Excerpt from Rana's map module. Error checks are performed to ensure that the radius argument is of type number and bigger than 0.

Appendix B

The Greenfield Model

This is the full agent design for the adapted Greenfield behavioural model (from page 14). Complete with data collection capability. Will write oscillation data to a comma-separated file, to enable plotting in Rana using a third party program, such as gnumeric [72].

```
-- data sets
Olevels = {}
dataFactor = 100
step = 0
iteration = 1
-- Oscillator values:
T = 0.500 -- time period.
e = 0.030 -- period variance with mean of 0.
r = 0.100 -- falltime.
Tt = 0 -- active period targeted time.
Tn = 0 -- active period time
y = 0.05 -- interrupt pause
s = .2 -- the PRC slop value.
t = 0.060
x = 0.050
yy = 0
pause = false
reset = false
peaked = false
--Import Rana lua libraries.
Event   = require "ranalib_event"
```

V. Appendix

```
Core    = require "ranalib_core"
Stat    = require "ranalib_statistic"
    function initializeAgent()
        Tt = T + Stat.randomMean(e,0)

        if ID==2 then
            PositionX=10
            PositionY=10
    elseif ID==3 then
            PositionX=20
            PositionY=10
    end
end
```

Listing B.1: Setup values and the initialization function for the adapted Greenfield agent

```
function TakeStep()
    Tn = Tn + STEP_RESOLUTION

    if pause == true and Tn >= yy then
        table.insert(Olevels, Core.time()..","..0)
        pause = false
    end
    if peaked == false and Tn >= Tt-t then
        table.insert(Olevels, Core.time()..","..1 ..",peak")
        peaked = true
    end
    if reset==false and Tn >= r then
        table.insert(Olevels,Core.time()..","..0)
        reset = true
    end
    if Tn >= Tt then
        Event.emit{description="Signal"}
        table.insert(Olevels, Core.time()..","..(Tn-r)/(Tt) ..",call")
        Tt = T + Stat.randomMean(e, 0)
        Tn = 0
        peaked = false
        reset = false
    end
end
```

Listing B.2: The take step funtion for the adapted Greenfield agent

```
function HandleEvent(event)
    if Tn >= x then
        -- write data to the Olevel table:
        table.insert(Olevels, Core.time()..","..Tn/Tt..",interrupt")
        table.insert(Olevels, Core.time()..","..  0)
        --calculate new period
        Tt = Tn * s + Tt
        yy = y+Tn
        if peaked == true then
        Tn = Tt-y
        end
        pause = true
    end
end
```

Listing B.3: The handle event function for the adapted Greenfield agent

Appendix C

Rana Demonstration Models

The models accompanying the Rana distribution[1] are the following:

1. ping-pong: the demonstration agent used in the event handling example in the Rana demonstration chapter on page 82.
2. data-collection: the three agents that comprise the demonstration on data collection in the Rana demonstration chapter on page 87.
3. painter: a single agent that on every take-step phase paints a pixel on the map a random colour.
4. value-sharing: a demonstrating of using the Rana API for sharing values.
5. bat: a number of agents that comprise the demonstration agents used in the event visualization demonstration on page 135.
6. move: a single agent that demonstrate agent movement by moving around at random.
7. repulser: the single agent that was used to demonstrate movement and collision detection in the Rana demonstration chapter on page 85.

[1]The newest versions can be found here: https://github.com/sojoe02/RANA/tree/master/lua_agents

8. flasher: a demonstration of agent manipulation, features a 'master-flasher' that adds agents at random intervals, the agents change to a new random colour on every take-step phase, creating a disco effect.

9. radial-scanner: a single agent that demonstrate simple radial map surveying and collision detection.

10. targeted-ping-pong: demonstrates the the event groups by joining a random group from between 1 and 10, only members of its own group will register the pings it sends out.

11. foraging-frog: the model that was used in the Rana demonstration chapter on modular agent design on page 91. features a master agent for configuration of the simulation.

12. greenfield: contains all the agents used in the chapter describing the adaptation of the Greenfield model (page 167).

Appendix D

Testing

The following sections represents tests on Rana's precision and performance. The following tests has been run.

- **Event Propagation Precision**. Tests the precision level of the event propagation engine, for stationary agents.
- **Performance Tests**. Performance gain tests covering both multi-threading and LuaJIT versus regular Lua.

D.1 Event Propagation Precision

This precision test is a demonstration of the precision and limitations the precision level of Rana's event propagation engine.

D.1.1 Setup

The simulation consists of 4 agents. Stating from the first take-step phase agent 1 will emit an event to agent 2, which will respond by sending an event to agent 3, then agent 4, then agent 1 again and so fourth. The agent are placed in a square 100 by 100 metres.

Event propagation speed is equal to 400[m/s], with instant agent response agent 1 will receive an event every second 1[s].

Due to the mechanics of Rana's event-handling there is an imprecision caused by internal event processing delay which is 2 handle-event step precision level. one on event reception and one on event emission in the handle-event phase.

The experiment has been run 5 times with 4 simulation threads enabled, the where no variance in the output.

Figure D.1: Event propagation precision agent placement.

D.1.2 Results

The results of a simulation with run-time of 3700 seconds, handle-event precision level of 1×10^{-6} and take-step precision of 1×10^{-3}, are presented in table D.1.

D.2 LuaJIT and Multi-thread Performance

This is a two-pronged test, that demonstrates the multi-core performance gain with LuaJIT [56] versus regular Lua [31]. The test system and software versions are listed in table D.2. To avoid operating system based context switching,

Metres(m)	Core Time(s)	ExpectedTime(s)	AdjustedTime(s)
400	1.000008	1	1
16,000	40.00032	40	40
160,000	400.0032	400	400
480,000	1200.0096	1200	1200
960,000	2400.0192	2400	2400
1,440,000	3600.0288	3600	3600

Table D.1: First column is the amount of metres the events have propagated. Second column is the time registered time in the simulation by agent 1 on event reception. Third is the expected simulation core time. Fourth column shows the event propagation time when intra-agent delay has been removed.

affecting performance measurements, as few as possible user-space programs where left running during testing.

Cpu	Intel i7-4790K, 4 Cores, 8 threads
Memory	16 Gb, DDR3
Operating System	Arch Linux [79], Kernel 4.5.4-1-vfio
Qt version	5.7.0
LuaJIT version	2.0.4
Lua version	5.1

Table D.2: System specifications for performance tests.

Two agent designs has been tested: an event heavy ping-pong with simplistic agent behaviour, and an event light simulation with complex agent behaviour.

All simulations have a runtime of 100[s], handle-event precision level of 1×10^{-6} and take-step precision of 1×10^{-3}.

D.2.1 Ping pong Test

This first test uses a ping pong agent design like the one presented in the Rana demonstration chapter (section 5.2.1 on page 5.2.1).

100 agents are randomly distributed across a 200 by 200 metre map, each simulation is run 5 times, as there has been no noticable variance the average runtime is presented.

D.2.1.1 Results

The results of the test is presented in table D.3.

Lua engine	1 thread	2 threads	4 threads
Regular Lua	166[s]	158[s]	146[s]
LuaJIT	128[s]	119[s]	109[s]

Table D.3: Shows the average runtime for both regular Lua and LuaJIT, with 1,2 and 4 threads.

The (rounded) average of events emitted is 1,000,000 with 20,000,000 event references, which means that this is a very event heavy simulation. Since event-handling is centralized the performance gain from multi-threading is not too significant. LuaJIT's improved C++ interface is clearly better than regular Lua despite the fact that there are no compile targets for the JIT engine within the agent code.

From slowest,regular Lua on 1 core to fastest, LuaJIT on 4 cores there is a 52% runtime difference.

D.2.2 Foraging agent results

This first test uses a modular foraging agent design like the one presented in the Rana demonstration chapter (section 5.3 on page 5.3). This simulation features 300 active agents, 100 of each population.

D.2.2.1 Results

The average runtime results of the simulations are presented in table D.4.

Lua engine	1 thread	2 threads	4 threads
Regular Lua	60[s]	40[s]	31[s]
LuaJIT	43[s]	30[s]	23[s]

Table D.4: Shows the average runtime for both regular Lua and LuaJIT, with 1, 2 and 4 threads.

This simulation show better performance improvement both with LuaJIT and multi-threading. This is a more distributed simulation than the ping-pong one, as it has more active agents, fewer events emitted and more complex agent behaviour.

Runtime difference from slowest, regular Lua on 1 core to LuaJIT on 4 cores is 160%, which is a significant performance gain.

D.2.3 Further Performance Testing with LuaJIT

Neither of the two tested models represent calculation heavy models, so for further experimentation we have implemented a function with tail recursion (listing D.1). The test will reuse the ping-pong agent again, the only change is that it will evaluate the 1000th fibonacci number on every take-step phase to simulate more computer heavy behaviour.

The mode and testing scenario is the same as for the ping-pong performance test.

D.2.4 Results

The performance results are listed in table D.5.

As the behaviour of the ping-pong agent remain the same, the simulation is still very event heavy with an average of 1 million events, and 20 million references.

```
function fibonacci(n)
    local function f(a, b, n)
        if n < 3 then
    return b
    else
        return f(b, a+b, n-1)
    end
end
    return f(1,1,n)
end
```

Listing D.1: Lua fibonacci function with tail recursion

Lua engine	1 thread	4 threads
Regular Lua	490[s]	306[s]
LuaJIT	179[s]	135[s]

Table D.5: Shows the average runtime for both regular Lua and LuaJIT, with 1 and 4 threads.

Runtime difference from slowest, regular Lua on 1 core to LuaJIT on 4 cores is 363%, which is a substancial gain. The runtime difference between the fastest regular Lua to the fastest LuaJIT test is 227%.

D.3 Conclusion

Rana is developed with very high precision event propagation in mind and the first test confirms that there is no precision degradation event on longer simulations with the demonstrated event juggling going on, with stationary agents.

The performance gain from using LuaJIT with complex behaviours, even with event heavy agent designs, is significant. Furthermore there was good multi-core performance gain despite heavy event interaction (which is sequential).

Appendix E

Available Lua Module and C++ functions

This chapter contains adapted chapters of the Rana wiki [1] sections detailing the functionality of the included Lua modules and the Rana core API functions. These functions are all included with default Rana distribution. The github wiki pages have been converted to Latex format using the Pandoc [46] document conversion tool

E.1 Module Overview

This is an overview of the modules provided with Rana.

E.1.1 Agent

Agent manipulation module, that allows for adding and removing agents and joining event groups (module name `ranalib_agent`).

[1] The Rana Wiki is part of the Rana github development page https://github.com/sojoe02/RANA/wiki

Function	Arguments	Description
addAgent	`path, positionX, positionY`	Adds an agent, with the given path, the path is relative. If no position X or Y is given a random position will set for the new agent. Returns the ID of the new agent
removeAgent	`ID`	Removes the agent with the given `ID` if it exists. Returns true if successful
joinGroup	`groupID`	Add this agent to an event group, if the group does not exist a new one will be made
leaveGroup	`groupID`	remove this agent from an event group. Returns true if successful and false if the agent is not member of that group
setStepMultiple	`multiple`	Defines the `StepMultiple` variable with error checks. Returns true if successful.
changeColor	`{id=ID, r=0, g=0, b=0, alpha=255}`	changes the color of an agent e.g `Agent.changeColor{r=255, g=123}`, returns a boolean denoting whether the color values are valid or not

E.1.2 Collision detection

Provides the ability to check positions for the presence of agents via a map-wide collision grid. Collision detection is handled by a position mapping implementation separate from the Rana core that otherwise handles movement. Only an agent that sets the `GridMovement` boolean to `true` either during it's initialization phase or when moving will be registered in the collision grid.

As agent positions are floating point values with 64 bit precision, the grid needs to be defined at a certain precision level. At scale 1 the grid considers all agents within 1 meter on the same spot. E.g one agent on `1.323,1.321` and another on `1.9,1.732` will be considered to be on the same spot. It is possible to control the scale of the grid by re-initializing the grid with a different scale.

Function	Arguments	Description
checkPosition	`x, y`	checks whether there are any agent on `x,y` and returns a table with agent IDs present on the `x,y` position. If the agent checks it's own position it will get it's own Id back along with other agents present
checkCollision	`x, y`	checks for collision on x,y return `true` if there is one
addPosition	`x, y, ID`	Adds a position to the collision grid on position x,y with ID
updatePosition	`newX, newY, ID`	Updates a position on the map, this is not needed for standard collision detection, as an agents positions on the collision grid is automatically updated if `GridMovement` is set to true
updatePosition-IfFree	`newX, newY, ID`	Updates a position on the map, if that position is free of other agents. This is handled atomically across execution threads
reinitializeGrid	`scale`	Sets the precision of the collision grid, default is 1[m]. Scale is defined in meters. This will clear all existing positions

Function	Arguments	Description
radial-CollisionScan	`radius`	Very fast radial scanning via an API call, will return a nested table with all ID's and their position X and Y within range of the agent in a given radius. Unlike checkCollision this function will not return the agent itself, e.g `{[1]={posX=161, posY=146, id=52 }, [2]={posX=174, posY=147, id=21 }}`. Returns `nil` if no agents are found. for an example see agent `09_radial_scanner.lua`.

E.1.3 Core

Provides core functionality, such as retrieval of current time (module name `ranalib_core`).

Function	Arguments	Description
stopSimulation		Stops the simulation and runs `cleanUp` for all the agents
time		returns the current time of the simulation in seconds(64bit precision)

E.1.4 Event

Module for event distribution (Module name `ranalib_event`).

Function	Arguments	Description
emit	optionTable `{speed, description="event", table, targetID and targetGroup}`	Emission of events with dynamic arguments. E.g `Event.emit{speed=343, targetID=1, description="foo"}`

The table argument can either be a string representation of a Lua table or a lua Table. To optimize performance it is best to initialize event tables if simulation performance is important.

E.1.5 Map

Provides map interaction and scanning capability to the agent. These functions allows the agent to read and write to the 2D map in Rana [2].

Function	Arguments	Description
checkColor	`x, y`	Checks the color of the given x,y coordinate, returns a table with the R,G and B values
modifyColor	`x, y, r, g, b`	Changes the color of a given `x,y` coordinate. `r,g,b` are channel red greed and blue respectively, they are 8 bit values and must be between 0 and 255.

[2] https://github.com/sojoe02/RANA/wiki/map-manipulation

Function	Arguments	Description
getRadialMask	`radius`	returns a table with valid coordinates within the given radius, uses the same fast algorithm as the radial collision detection algorithm, `09_radial_scanner.lua` also displays this feature

E.1.6 Shared

Allows agents to share values of type number, table or string in central registers using a key, tables and strings share the same register (module name `ranalib_shared`).

Function	Arguments	Description
storeTable	`key`, `table`, `check`	Adds a lua table to the register, `check` is a boolean that if true will ensure that various error checks are performed before submitting a value to the register (default is `false`)
getTable	`key`	Retrieves a table from the register, if it exists
storeString	`key`, `string`, `check`	Adds a string table to the register
getString	`key`	Retrieves a string from the central register if it exists
storeNumber	`key`, `number`, `check`	Adds a number to the central number register

E.1.7 Statistics

Provides a number of rudimentary function to enable probabilistic agent functionality.

Function	Arguments	Description
randomInteger	`int1`, `int2`	returns a random non-floating point value of [`int1`,`int2`]
randomFloat	`float1`, `float2`	returns a random floating point value of [`float1`,`float2`[
randomMean	`deviation`, `mean`	returns a random floating point value with the mean and deviation provided
roundToStep	`value`	rounds a random floating point number to the step precision of the current simulation

E.2 API functions

Below is a listing of all the API functions available in Rana.

E.2.1 Output

These functions have are designed to be called directly by the agent, no module is needed.

Function	Arguments	Description
l_print	`string`	Prints `string` a html formatted string to simulation output.
l_debug	`string`	Same as `l_print`, though the output of this can be disabled in the menu
say	`string`	see `l_debug`
shout	`string`	see `l_print`

E.2.2 Map

Function	Arguments	Description
l_modifyMap	`x, y, R, G, B`	Changes the color of the map, at `x,y`, where `R, G` and `B` corresponds to a signed 8 bit integer, red, green and blue
l_checkMap	`x,y`	Returns the 8 bit signed(0-255) R,G and B values of a specific section of the map. Will return 256,256,256 if the position checked is out of bounds.

Function	Arguments	Description
l_radial-MapScan	`radius, x, y, R,G,B`	Returns a nested table with valid `x,y` coordinates matching the colour argument given. `radius`
l_checkAnd-Change	`x, y, color1, color2,`	Checks for a colour value matching colour1 on the map, if it matches it will change it to colour2. `radius`

E.2.3 Shared Values

Function	Arguments	Description
l_addSharedNumber	`key, number`	adds `number`(64 bit float) to a shared hash-map, indexed by `key` of type string
l_getSharedNumber	`key`	returns the number associated with `key`, if `key` does not exist it returns "no_value"
l_addSharedString	`key, string`	adds `string` to a shared hash-map, with `key` of type string. E.g. this can be used to store serialized tables
l_getSharedString	`key`	returns the string associated with `key`, if `key` does not exist it returns "no_value"

E.2.4 Physics

Function	Arguments	Description
l_speedOfSound	`myX`, `myY`, `origX`, `origY`, `propspeed`	Calculates the arrival microstep, for something that propagates from `origX,origY` to `myX,myY` with the speed of `propspeed` (m/s)
l_distance	`myX, myY, origX, origY`	Calculates the amount of units between `myX,myY` and `origX,origY`
l_getRandomFloat	`float1, float2`	Returns a 64 bit float between [`float1,float2`[, uses Mersenne twister with a simulation central seed
l_getRandomInteger	`uint1, uint2`	Returns a 64 unsigned integer between [`uint1, uint2`], uses Mersenne twister with a simulation central seed

E.2.5 Simulation Variables

Function	Arguments	Description
l_currentTime		Returns the current microstep.
l_getMacroFactor		Returns the macrofactor of the simulator
l_getTimeResolution		Returns the microresolution
l_getEnvironmentSize		Returns width and height of the environment(starts at 0)
l_getAgentPath		Returns two strings, the path of the agent(no filename) and the filename of the main lua agent file

E.2.6 Collision Detection

Function	Arguments	Description
l_initializeGrid	`scale`	Reinitializes the collision grid, with a new scale, clears the existing one of all data
l_addPosition	`x`, `y`, `ID`	Adds an `x`,`y` position to the collision table, with an agent `ID`. This can be used multiple times to occupy more than one square.
l_updatePosition	`oldX`, `oldY`, `newX`, `newY`, `ID`	Updates a position from `oldX`,`oldY` to `newX`,`newY` , if the agent has *GridMove* set to `true` it's positions will automatically be updated in the collision table
l_updatePosition-IfFree	`oldX`, `oldY`, `newX`, `newY`, `ID`	Updates a position in the collision detection map, if the position is free, this is done *atomically* across execution threads
l_checkPosition	`x`, `y`	Returns a list of the ID's of the agents at position `x`,`y`
l_checkCollision	`x`, `y`	Returns a boolean that is true if an agent is occupying `x`,`y`
l_radialCollision-Scan	`ID,radius, x, y`	Returns a nested table containing all the agent ID's within the radius along with the agents positions, will ignore agents with id=ID

E.2.7 Simulation Manipulation

Function	Arguments	Description
l_stopSimulation		Tells the simulation core to stop the current simulation when the next macroStep is done

E.2.8 Agent Manipulation

Function	Arguments	Description
l_addAgent	`x`, `y`, `z`, `path`, `filename`	Adds a new agent at a given `x`,`y` and `z` position using `path` and `filename`, the simulation will stop with a warning if the agent source cannot be found. Returns the id of the new agent
l_removeAgent	`id`	Removes agent with *id*. Returns true or false depending on whether removal is successful. This will also clear the Agent from the collision table
l_changeAgent-Color	`id`, `r`,`g` ,`b` ,`alpha`	changes the color of the graphic representation of the agent, returns `true` if the color values are valid

List of Figures

3.1 Module and API dependency diagram. 37
3.2 Rana design structure diagram. 44

4.1 Sequence of the event as it travels from the agent, through the API and core and into the event handler 55
4.2 Sequence of how the event data is submitted to all agents, which then calculate an event arrival time, and generate an event reference 55
4.3 The handling of an event reference to distribute event data to a single frog responder agent 57
4.4 Illustration of the Lua engine and agent interface. 60
4.5 Classes and structure of the User Interface. 66
4.6 Classes and structure of the Core 69
4.7 Classes and structure of the Agent Domain. 70
4.8 Classes and structure of the API. 71
4.9 Illustration of the take-step threading phase. 73

5.1 The Simulation control interface 79
5.2 The live simulation view 81
5.3 Sample output and placement of a ping pong agent simulation. . . 85
5.4 The behaviour states of the repulser agents take step phase. . . . 86
5.5 Output of a simulation, with 100 repulser agents. 87

List of Figures

5.6 States of the relevant phases of the master agent. 88
5.7 The main states of the relevant oscillator phases 89
5.8 The oscillation values for each individual oscillator in a 10 second simulation. 90
5.9 The main statemachine of the foraging frog agent. 92
5.10 The foraging frog's calling statemachine. 93
5.11 The foraging frogs foraging statemachine. 94
5.12 Data collector agent output for the foraging frog model. 97

7.1 Illustration of the event-map, z-blocks are illustrated as coloured squares. Each different colour represents a different period in time. 107
7.2 Event visualizer design diagram. 111

9.1 The event-processing panel for the visualization user interface. . . 129
9.2 The event map panel, for the event visualization user interface. . 130
9.3 The handle event and its sub-state for evaluating the neighbouring emitters 134
9.4 Oscillation plots for the event-processing oscillator. 135
9.5 The drone demonstration for event-visualization set up. 136
9.6 The visualization of the intensity of events at different times. The setting of the intensity display type is additive. 138
9.7 The visualization of the intensity of events at different times. The setting of the intensity display type is average. 138

12.1 An example of a map section that has been converted to the Rana map format. 156

13.1 The mining patterns exhibit in the mining robot experiment. . . . 164
13.2 The mining robot performance experiment results plot 164

14.1 Freerunning oscillator 171
14.2 Two inhibing oscillators without PRC 174
14.3 Two inhibiting oscilattors with PRC 176

14.4 Box-plot synchronization performance graph with two inhibiting agents with PRC 178
14.5 Box-plot synchronization performance graph for different PRC values, accompanied by a single free-running oscillator. 179
14.6 Box-plot graph of total amount of calls for different PRC values, accompanied by a single free-running oscillator. 180

D.1 Event propagation precision agent placement. 226

VI

Publications

"Besides", he said to himself, *"I have imagined too many possibilities for tragedy, and it is only proper to melancholics to generate spectres that reality is unable to imitate."*

The Island of the Day Before, Umberto Eco

Biomimetic Agent Based Modelling Using Male Frog Calling Behaviour as a Case Study

Søren V. Jørgensen[1], Yves Demazeau[2], Jakob Christensen-Dalsgaard[3] and John Hallam[4]

[1] Center for Biorobotics, Maersk McKinney Moeller Institute
University of Southern Denmark
svjo@mmmi.sdu.dk
[2] CNRS — LIG
38000 Grenoble
Yves.Demazeau@imag.fr
[3] Center for Sound Communication, Institute of Biology
University of Southern Denmark
jcd@biology.sdu.dk
[4] Center for Biorobotics, Maersk Mc-Kinney Moeller Institute
University of Southern Denmark
john@mmmi.sdu.dk

Abstract. A new agent-based modelling tool has been developed to allow the modelling of populations of individuals whose interactions are characterised by tightly timed dynamics. The tool was developed to model male frog calling dynamics, to facilitate research into what local rules may be employed by individuals to generate their observed population behaviour. A number of existing agent-modelling frameworks are considered, but none have the ability to handle large numbers of time-dependent event-generating agents; hence the construction of a new tool, RANA. The calling behaviour of the Puerto Rican Tree Frog, *E. coqui*, is implemented as a case study for the presentation and discussion of the tool, and results from this model are presented. RANA, in its present stage of development, is shown to be able to handle the problem of modelling calling frogs, and several fruitful extensions are proposed and motivated.

1 Introduction

In many cases, modelling interaction of agents in a population at the level of the agents themselves requires an ability to manage timing constraints. For example, calling frogs emit their calls at times which are influenced by what they hear, and the time at which they hear emitted calls from other frogs are determined by the physical process of sound propagation in their environment. In the current state of the art there is a lack of agent-based simulation tools able to support such precise time-based models of large populations of agents; we therefore describe RANA, a new tool we have built with that goal in mind.

Agent-based modelling offers an interesting way of performing biologically inspired simulations. Agent-based social interaction models have been constructed[3], and biomimetic modelling is supported by various tools including the Netlogo framework[15], which contains a good number of ready-to-run biomimetic models. An example of biomimetic agent-based behaviour modelling has been published on Caribou herds in the Arctic[9]. However, simulating male frog calling behaviour dynamics has a different set of requirements from agent-based models such as these: male frog calling simulation requires the ability to perform simulation of high-precision timing-based emission and processing of events, taking physical constraints such as the speed of sound and neural processing time into account. What is proposed here is an agent modelling framework that is flexible and powerful enough to enable both advanced behavioural design and the precision required to achieve results from simulated agents consistent with observation of the natural creatures. Our solution offers flexible agent implementation using an existing high-performance scripting language[5], which interfaces with a user configurable event processing framework — events in this case being any external action taken by the agents in the simulation, such as calling out or moving.

The main purpose for the suggested agent modelling framework is to enable the design of agent behaviour that mimics observed behaviour in natural agents, so as to further understanding of the natural agents' interactions and how group behaviour might be affected by individual agent attributes, while taking physical constraints such as neural processing time and sound propagation into account.

2 Frogs as a Case Study for Agent Based Modelling

Male frog mating call dynamics is a complex subject. Frogs have through evolution been physically shaped to optimise their chance of survival and procreation through highly-specialised calling behaviour. The evolution of each subspecies of frog is strongly influenced by the success of male frogs in attracting females while using minimal energy, in the presence of competing calls from other males and interference from other environmental factors. It is not just a matter of optimising the call strength, duration and frequency of the individual male frog. Different species employ different algorithms that take both physical and environmental constraints into account when choosing when to emit a call: for instance, a poorly-timed call might alert predators and enable them to locate the unlucky individual.

It is worth mentioning that there are generally two types of calling behaviour, antiphonal (asynchronous) and chorusing (synchronous). Natterjacks chorus, for example while *E. coqui* perform their calls asynchronously with respect to two or three neighboring callers.

It would be interesting to uncover what external attributes a male frog can consider and use in order to achieve optimal performance in the highly competitive environment during mating season, and how the choices of the individuals affect the dynamics of the whole frog population. To enable the simulation of male

calling dynamics a high-performance agent-based modelling tool is warranted; a tool that allows for very high precision, optimised for event broadcasting rather than peer-to-peer interaction.

Simulating the Asynchronous Calling Behaviour of *E. coqui* Simulations of the Puerto Rican Tree Frog *E. coqui* have been described in the literature[2]. In that project, simulations were constructed based on models of two or three frogs. The project also determined that each male *E. coqui* only reacts to a maximum of three neighbouring callers.

Only involving a couple of individuals in a simulation is questionable since *E. coqui* populations generally have a very high density of male individuals per acre (up to 133,000 [8]). It would be interesting to simulate a much larger population to check how population size and density might affect each individual's behaviour.

Population Wide Stochastic Modelling Previous work modelling whole populations has taken the approach of stochastic modelling [10], which allows the setup of an efficient population-wide model that can take several attributes into account as well as the different states the male frog can be in during mating season. The model deals with the four different states listed below.

- **Calling**: the frog emits calls and expends energy doing so.
- **Foraging**: the frog does not call but charges up energy instead.
- **Satellite**: the frog does not call, but attempts to intercept females attracted by nearby male callers. This is typically a behaviour used by weaker males, where they attempt to save energy by using the call of another male frog to attract mates.
- **Hiding**: the frog can neither mate nor recharge energy.

The only stochastic variable in the model is the rate at which the frog's energy level replenishes.

Neural Network Decision Model An alternative approach to the problem of modelling the frog's decision process is to use a neural network[11] where relevant attributes both dynamic and static — such as refractory period, energy level and neighbouring call strength — are presented as inputs to a network that determines the frog's actions.

A neural network approach has been successfully implemented for a related case: to determine female frog response biases to male frog calling[13], of the *Túngara*. The neural network managed, successfully, to recognise the males' mating calls and could, with a great degree of precision, determine how well females generalised to many novel calls.

3 Utilization of Agent-Based Simulation to Improve the State of the Art

The existing modelling techniques mentioned above allow the construction of rather advanced simulations; however, the models are either limited in scope or do not take individualism directly into account. Agent-based modelling is flexible in that agents are modelled as individuals, and may all differ. Their decisions are based on locally available information, so can reflect rather more specific or sophisticated modelling assumptions than population-based models; for instance, multi-species interactions can easily be handled in an individual-based framework. It becomes possible to experiment with the relationship between locally-available information, local decisions and emergent global behaviour of the agent population in a wider range of circumstances than population models typically allow. Agent-based modelling is, depending on the modelling language, very flexible and is agnostic about the implementation of the local decision model, enabling for instance neural network representations or state-based models that describe an agent's behavioural profile. Advanced swarm behaviour achieved through agent-based modelling has previously demonstrated in Reynolds' graphic behavioural model[14].

3.1 State of the Art Modelling Frameworks

Several existing agent-based modelling frameworks were considered for frog calling behaviour simulation, to determine whether they were usable as-is, or with reasonable extension and modification. The following frameworks were reviewed.

- **NetLogo**: Even though there exist many examples of Netlogo models of biomimetic behaviour, the Nlogo modelling language is limited in scope, and Nlogo is not designed for high-precision time-based simulation.
- **Repast[12]**, in its various configurations, supports Nlogo, Java and C++ agents. However, modelling options such as C++ and Java, necessary for time-based event management, will require framework recompilation.
- **GAMA[4]**: The GAML modelling language is limited in scope, making the necessary modelling primitives hard to realise, and GAMA is tied to the heavy Eclipse development environment.
- **JADE[1]**, Java based agent design requires sophisticated programming knowledge. It supports a very limited number of agents, and is not suited for high-precision simulations.

While it might have been possible to adapt one of the frameworks to suit the problem domain, none of them naturally or natively supports the simulation of large numbers of agents interacting via events whose timing is critical and determined by environmental physics. We therefore decided to develop a new framework from scratch. This enables the creation of a platform-independent framework specifically designed towards high-volume high-precision simulations,

thereby offering good performance on agent handling without compromising agent design complexity or simulation integrity.

It also gives the opportunity to develop a lightweight framework, which at its core is independent of heavy components such as Eclipse and the Oracle Java Virtual Machine.

3.2 Agent Design Using Lua

The developed simulation framework, RANA[6] (Reproduction of Artificial and Natural Agents), uses Lua[5] as its principal agent modelling language, though models can also be coded in C++. Instead of inventing a new agent design language, Lua is chosen as the main modelling language. Lua is widely used as a scripting language for embedded systems, and is compact, clean, efficient and easy to interface with lower level languages such as C or C++. Furthermore, it is a powerful modern language with a clean design, making it simple to learn. It has a reasonable set of libraries available for use in the models, and is easy to interface to new libraries should that be needed.

Using Lua makes it possible to offer several flexible template agents that regular users can utilise to design their own agents while allowing advanced users to design and implement complex agent behaviour. The only constraint on models is that they implement the predefined functions for event handling and processing through which the event-handling core of the simulation interacts with the models.

Advanced users can choose to implement their agent designs in C++, and compile the agents into the framework. Compilation time is not significant on modern platforms since RANA is a lightweight framework, and C++ agents carry a significant performance advantage, but the implementation complexity is significantly greater for a given behaviour complexity than constructing the model using Lua.

By providing a number of template agents the modelling threshold is set to a point where it should be possible for non-programmers to design complex biomimetic interactive behaviour simulations. There is no requirement to run a heavy duty development environment, or have knowledge of software development tools: a simple text editor is all that is needed. Nevertheless Lua enables the design of advanced agents with attributes and functions that can help the researcher uncover the underlying mechanics of the simulated agents through careful experimentation. Furthermore it is possible to reach a very high level of abstraction through the development of species-specific Lua libraries.

3.3 Agent Mechanics

The RANA agent has several functions available to it during a simulation run. Mainly there are two different event-handling functions, one for handling events issued by fellow agents and one that allows the agent to perform an internal action at a clearly defined time. The agent also has an initialisation function that will be called on simulation start-up, where it can set up its behavioural

attributes. There is also a simulation-end function which allows the agent one final action enabling (for example) agent-specific data collection.

At the start of a simulation run each agent is initialised. The simulator then moves time forward until an agent decides that it is time to initiate an event. Fellow agents register the event and calculate when it will arrive locally, taking into account environmental physics, i.e. a call event will propagate with the speed of sound. The agent can also take neural processing time and other internal factors into account when responding to the event, and thus a group dynamic begins, where agents respond to each others' events depending on their internal and external state. The agents can then output the results of their individual behaviour to the console or write it to a binary data file which can be used for post processing.

4 Model Testing

Example models have been designed in an attempt to build agents that mimic behaviours described in the aforementioned literature on frog modelling.

4.1 The *E. coqui* Male Calling Behaviour

The modelled *E. coqui* has, aside from position and an ID provided by the simulator, the attributes listed in Table 1.

Table 1. Attributes of the template frog and their distribution

Attribute (value)	Description
intensityThr (0.8)	Minimum Sound intensity to trigger response, intensity level at source is 1.0
CallStrength (35–50)	How far a call travels before reaching intensity level 0.5
EnergyLvl (0.0–1.0)	Amount of energy a frog starts out with
EnergyRegenRate (0.01–0.0125)	Amount of energy a frog regenerates every second
CompelRegenRate (0.02–0.04)	If the frog has not registered any neighbouring calls for a while it will perform a spontaneous call once the Compellevel goes to 11. The CompelRegenRate determines how much the compel level regenerates every second, and any call made will reset the compel level. This attribute determines how bold the frog is.

The model will actively attempt to avoid call overlap with its two strongest neighbours, which is a normal *E. coqui* male behavioural pattern, taking on an

asynchronous behaviour. The duration of the *E. coqui* call is 375[ms]; the model takes this into account, as well as the frog's neural processing time of approx. 35[ms].

The frogs are placed in a grid, with equal distance to one another.

The distribution function for the call events sound intensity is defined as an exponential decreasing function, which outputs a sound intensity level between 0 and 1 as a function of distance from origin. The frogs in the simulation will ignore any intensity level below 0.8 (calls made more than 6 to 8[m] away).

The model registers, via a special data-collector agent, how many calls it has made with no neighbouring overlap, with overlap from a single neighbour and how many calls it has attempted to make with no overlap.

The simulation area size is 10x10[m], and three simulations with increasing density is run, with 15, 35 and 63 frogs. Each simulation is run 5 times.

Results In a sparse environment with 15 frogs, all frogs managed to expend their energy as fast as it regenerated. Thus they performed optimally from an energy point of view. 82–87% of all calls were made asynchronously with the chosen neighbouring callers; the rest were performed with overlap. The typical frog managed to anticipate when neighbouring callers would call in more than 98% of the cases.

With 35 frogs, the average percentage of success rate is 80–82%, the call frequency falls as a consequence of the density increase, the anticipation rate remains at 98%.

At 63 frogs, the average success rate is 79–80%, again the call frequency falls slightly, anticipation rate stays above 98%.

Discussion We observed that the proportion of calls overlapping with a chosen neighbour changes slightly with increasing density of frogs, for the range of densities tested. It was hoped that over time the model would cause the frogs to settle into a regular rhythm, and they did manage that reasonably well. The model did not achieve complete antiphonal success though. The reason for this is probably that while one frog might consider another a neighbour, the reverse is not necessarily true due to the variance in call strength.

Since agents were not able to move in this simulation, the individuals were forced to remain where initially placed, rather than being able to move to a more favourable position as a real frog might. The implementation of the states identified in the stochastic model could also give a more dynamic calling environment, and possibly decrease the overall call overlap.

It is worth noting the simulation results are heavily dependent on how likely the frogs are to initiate a call spontaneously and how advanced their call anticipation algorithm really is. Further observation into *E. coqui* calling behaviour could give a more precise view of how bold these frogs generally are.

4.2 Modelling Female Mating Call Response

For some species of frog females will perform courting calls[16]. These calls serve to entice nearby males to perform courting calls of their own. In this simulation we designed a female frog that moves around the environment at random. Whenever she comes within the vicinity of a strong male caller, as determined by the local sound intensity, she will begin responding with a courting call. This will cause every male in the female's vicinity to increase his call frequency and intensity. After a couple of seconds of courting, the female then moves to another location, looking for another group of males to entice.

Results The movement and calling of the frogs in this simulation is visualised via a separate piece of software developed to enable visualisation of the agent event activity[7]. The visualiser showed that the model was successful: the female's courting call will incite nearby males and they will then increase their call frequency either until the female stops courting or they run out of energy. Once the female stops courting, the affected males fall silent while their energy replenishes. Once replenished the males resume with regular mating calls.

Discussion Simulating movement and its effect on results is something that still needs to be researched properly. This simulation merely demonstrates that it is something the simulation core will support. However more advanced models will need to be designed in order to achieve proper results that are applicable to real behaviour of the simulated animal.

5 Further Work

There are also many possibilities for further study of the simple frog model described above: a full investigation of the effects of the various parameters on the population behaviour is the most pressing. Extension of the decision model to improve its abilities to desynchronise with its neighbours is also indicated. For instance, one might include a random back-off interval after a call, as is used in algorithms for coordinating access to shared media such as Ethernet links or wireless spectrum.

Performing biomimetic modelling has proven to be no simple task. The *E. coqui* model's relatively simple behaviour requires several hundred lines of script coding. If the tool is to be used more generally by non-programmers, a lot of the functionality needed to make a model must be implemented in a library of building blocks of some kind. Fortunately, Lua allows several straightforward ways to accomplishing this. Alternatively, a domain-specific modelling language could be devised, though by its very nature this restricts the freedom of the modeller while simplifying the expression of models that are supported by the language.

There are also many possibilities for expansion of framework, both in regards to model features and expansion of the simulation core. While the modelling capabilities meet our basic requirements of scalability to large numbers of agents (tests with thousands of agents and hundreds of thousands of events have been run successfully), there are some extensions that would strengthen the behavioural modelling capabilities of RANA.

The frog taxonomy is very diverse, and the communication algorithms each sub-species employs varies greatly, due to environmental factors and the physiology of the animal. Unfortunately, documentation of frog communication *algorithms* is quite sparse, and to enable the modelling of individual agents it is important to perform species-specific field work which entails setting up listening arrays to localise individual frogs while providing data on call timing and call characteristics. Furthermore, unobtrusive observation on the habitat may uncover movement patterns of males and females and general behaviour patterns. Other aspects such as mating success rate of satellites versus callers could also be uncovered better. This fieldwork will allow improvements to be made to the basic model frogs as further significant factors influencing calling are identified and taken into account.

The agent should also be able to query relevant factors of its local environment, such as temperature, humidity and lighting level. To enable this the simulator should include some environment simulation; as a minimum, a map of a habitat could be loaded into the simulator along with some known environmental parameters. Frog models could them employ movement to find environmentally-favourable spots from which to call.

6 Conclusion

A new modelling tool, RANA, has been presented which supports high-volume and high-precision agent-based modelling in which environmental physics plays a significant part in the interaction of agents. The simulation core handles the exchange of events with micro-second timing precision, while agents employ models coded as Lua scripts to initiate, receive and respond to events. The tool was developed to support agent-based modelling of calling frogs and is illustrated with a simple simulation of the asynchronous calling behaviour of the Puerto Rican Tree Frog.

The simulator incorporates features that allow the modelling of individual decision processes dependent on local environmental factors and internal state as well as communication from other agents. As such, it can be used for other simulation tasks in addition to the frog-modelling which provided the primary motivation for the work. Multi-species simulations are straightforward, as each agent acts as an individual with its own decision model.

Several proof of concept simulations have been performed using RANA, such as the reported frog simulation that includes females traversing the environment inciting males with courting calls. Simulations of predation and elimination of species based on call performance have also been attempted.

References

1. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE. Wiley (2007), http://www.amazon.ca/Developing-Multi-Agent-Systems-Fabio-Bellifemine/dp/0470057475/sr=8-1/qid=1170365284/ref=sr_1_1/702-0885532-1303250?ie=UTF8&s=books
2. Brush, J.S., Narins, P.M.: Chorus dynamics of a netropical amphipian assemblage: comparison of computer simulation and natural behaviour. Animal Behaviour (1989)
3. Gilbert, N.: Computer simulation of social processes. Social Research Update 6 (1994)
4. Grignard, A., Taillandier, P., Gaudou, B., Vo, D., Huynh, N., Drogoul, A.: GAMA 1.6: Advancing the art of complex agent-based modeling and simulation. In: Boella, G., Elkind, E., Savarimuthu, B., Dignum, F., Purvis, M. (eds.) PRIMA 2013: Principles and Practice of Multi-Agent Systems, Lecture Notes in Computer Science, vol. 8291, pp. 117–131. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-44927-7_9
5. Ierusalimschy, R.: Lua 5.2 reference manual (2011–2013), http://www.lua.org/manual/5.2/, [Online; accessed 1-Feb-2014]
6. Joergensen, S.V., Demazeau, Y., Christensen-Dalsgaard, J., Hallam, J.: RANA, agent based real-time broadcasting simulation framework (2013-2014), https://github.com/sojoe02/RANA, [Online; accessed 24-Jan-2014]
7. Joergensen, S.V., Demazeau, Y., Christensen-Dalsgaard, J., Hallam, J.: RANA visualizer an agent event visualization framework (2013-2014), https://github.com/sojoe02/Kasterborous_Visualizer/, [Online; accessed 24-Jan-2014]
8. Kaiser, B.A., Burnett, K.M.: Economic impacts of e. coqui frogs in hawaii. 2006 Annual meeting, July 23-26, Long Beach, CA 21313, American Agricultural Economics Association (New Name 2008: Agricultural and Applied Economics Association) (2006), http://ideas.repec.org/p/ags/aaea06/21313.html
9. Lesins, G., Higuclu, K.: Agent based modelling of caribou environmental interactions in the Canadian arctic. The International Environmental Modelling and Software Society (2010)
10. Lucas, J.R., Howard, R.D., Palmer, J.G.: Callers and satellites: chorus behaviour in anurans as a stochastic dynamic game. Animal Behaviour (1993)
11. M, S.M.P.: History's Lessons, A Neural Network Approach to Receiver Biases and the Evolution of Communication, pp. 67–78. Smithsonian Instituion Press, Washington and London (2001)
12. North, M.J., Nicholson, C.T., Ozik, J., Tatara, E.R., Macal, C.M., Bragen, M., Sydelko, P.: Complex adaptive systems modeling with Repast Simphony (2013), http://link.springer.com/article/10.1186/2194-3206-1-3/fulltext.html, [Online; accessed 3-Feb-2014]
13. Phelps, S.M., Ryan, M.J.: Neural networks predict response biases of female túngara frogs. Proceedings. Biological sciences / The Royal Society 265(1393), 279–285 (Feb 1998), http://dx.doi.org/10.1098/rspb.1998.0293
14. Reynolds, C.W.: Flocks, herds and schools: A distributed behavioural model, in computer graphics. SIGGRAPH pp. 25–34 (1987)
15. Tisue, S., Wilensky, U.: Netlogo: A simple environment for modeling complexity. In: in International Conference on Complex Systems. pp. 16–21 (2004)
16. Wells, K.D., Swartz, J.J.: The Behavioral Ecology of Anuran Communication, pp. 44–89. Springer Handbook of Auditory Research (2006)

RANA, a Real-Time Multi-Agent System Simulator

Søren Vissing Jørgensen
Centre for BioRobotics, MMMI Institute
University of Southern Denmark
Email: svjo@mmmi.sdu.dk

Yves Demazeau
CNRS — LIG
38000 Grenoble
Email: Yves.Demazeau@imag.fr

John Hallam
Centre for BioRobotics, MMMI Institute
University of Southern Denmark
Email: john@mmmi.sdu.dk

***Abstract*—Multi-agent simulation of populations of acoustically-communicating animals presents challenges for existing MAS tools. We therefore present RANA, a new portable tool that enables large-scale simulation of precisely-timed models based on broadcast events. Agents are programmed in Lua, allowing for individualisation and abstraction while retaining efficiency. Events are managed by the C++ simulator core. Full run state can be recorded for post-processed visualisation or analysis.**

The new tool is demonstrated in three different cases: a mining robot simulation, which is purely action based; an agent-based setup that is verifies the high precision exhibited by RANAs simulation core; and a state-based firefly-like agent simulation that models real-time responses to fellow agents' signals, in which event propagation and reception affect the result of the simulation.

I. Introduction

Simulating animal communication using multi-agent systems (MAS) is an interesting proposition. Computer simulations allow the study of how individual traits affect group behaviour in detail unachievable by observation of the animals in their natural habitats. In the case of insects and frogs the interesting traits are exhibited by males during the mating season. For frogs specifically, analysis of their communicative behavior [1] reveals that their behaviour is strongly influenced by the timing of calls of nearby peers. Furthermore internal attributes, such as disposition towards peers and energy level, also represent important factors. By utilising the theory of life- and social science MAS [2], [3], one can design and run simulations where animals are modelled using agent-based programming paradigms, in which individual variation and local environmental conditions can easily be included. The simulations then allow one to investigate the relationship between individual characteristics and emergent group behaviour.

Our specific interest in animals that use acoustic signals for communication places some unusual requirements on the simulation framework. First, sound is a relatively *slow* signal, with propagation delays of the order of milliseconds in realistically-sized environments. These time delays are crucial in determining group behaviour for animals such as insects [4] and frogs [5], and are also of the same scale as the neural and muscular delays incurred when the animals hear, process and emit sounds. Modelling these timing properties with high precision and accuracy is a core requirement.

Second, sound is a *broadcast* signal sent indiscriminately to anyone within range. However, the signal that arrives at an agent, and its interpretation, depends on the propagation path and conditions between the sender and receiver as well as the characteristics of the signal itself. The simulation tool must therefore efficiently support broadcast to diverse agent populations.

Third, animal populations of interest are often dense. with tens to hundreds of thousands of individuals potentially participating in group behaviour. To enable simulations of high volume animals, such as certain species of insect and fireflies, the framework must efficiently support large numbers of diverse agents.

Finally, user-defined post-processing strengthens benchmarking and analysis of a simulation, something usually seen as an obstacle to using MAS for research [6].

Practically, the simulation framework should be portable across platforms and offer interfaces suitable not only for experienced programmers but also for animal modellers (who often have no interest in programming per se).

Unfortunately, as we see in a moment, state-of-the-art MAS platforms do not address these requirements well; hence the development of RANA, a new framework specialised toward this kind of modelling task.

II. MAS — State of the Art Analysis

We present brief descriptions of Netlogo, Repast and MASON, which we consider represent the current state of the art. A table listing relevant requirements facilitates comparison of the three frameworks in the context of our chosen tasks and indicates the need for RANA.

A. Netlogo, Repast and MASON

Netlogo [7] is a popular and proven open-source framework in Java, designed to perform social and life science MAS simulations. It can model social systems' development over time and its uses range range from modelling wealth distribution [8] to simulation of caribou herd behavior in the arctic [9]. It can be distributed, via Hubnet [10], to networked devices each representing one or several agents. Netlogo is designed to support agents whose behaviours create an emergent scenario which is to be observed during the simulation run. The suite supports different types of visualisation, such as plots, 2D and 3D model views and has user interactive controls such as a speed slider. It has some support for simulation data post-processing.

The Repast framework [11] exists in two different versions. The end user version, called Simphony, is Java-based and similar to Netlogo. It runs simulations on standard desktop computers. The second version is a *High Performance Computing* C++ based module designed to run high performance C++ agents in a super-computing environment. Simphony shares many of the traits of Netlogo and is even capable of running models designed in the Netlogo agent design language — Nlogo.

MASON [12] is a third Java-based simulation core with an attached visualisation library. Its primary application is the simulation of peer to peer communication creating observable and measurable emergent behavior. Its documentation states that it supports *"up to"* one million simulated agents, model detachment and check-pointing (which allows simulation states to be saved and resumed). As of 2015 Mason supports some form of paralization, but offers the model designer little help with this.

B. Feature-set comparison

Table I lists features that are important for real-time simulations as well as more general agent swarm simulations and compares the three state-of-the-art tools described above. The table comprises both necessary and useful features.

TABLE I
FEATURES NEEDED FOR SIMULATIONS OF ANIMAL ACOUSTIC COMMUNICATION

Feature	NetLogo	RePast	MASON
1. Micro Level Precision			
2. Event Broadcast Optimization			
3. Many Agents Possible	x	x	x
4. Visualisation	x	x	x
5. Multi-tier agent design	x	x	
6. Event and agent post-processing			
7. Multi-platform support	x	x	x

Descriptions of features listed in table I are provided below.

1 Event handling with precision at least, or better than, a real-time micro-second.
2 Optimisation for population-wide events, which may be relevant for several or many individuals.
3 Support for more than 10.000 unique individuals.
4 Live or post-processed view of agent behaviour.
5 Support for multiple agent design abstraction layers.
6 Events and their propagation can be restored with no loss.
7 Portable to Windows, Linux and Macintosh.

C. Discussion

Performing realistic animal communication simulations with true sound propagation requires precise modelling of real-time and broadcast events consistent with physical law. Taking account of animals' internal processing delays is also essential. It is clear from the table that the three frameworks are not specialised for this, lacking features 1 and 2.

It would probably be possible to extend one (or all) of the three to include the necessary features. By adopting a new approach to both design and implementation it is possible to diversify the offerings of MAS simulation tools while obtaining a tool designed specifically for a range of "real-time" multi-agent simulation tasks. We name this new framework RANA ('Reproduction of Natural and Artificial Agents'), an acronym inspired by the Latin moniker for frog (*Rana*).

III. THE RANA FRAMEWORK

RANA uses a client-server architecture to separate the simulation core and user interface. See figure 1 for an overview of the structure of the system. When run locally, a graphic client using Qt both provides a control interface to the simulator core and offers some data visualisation options. In remote mode, the user interface is textual, based on the Ncurses library — allowing control of simulations and an overview of progress while using, for instance, an SSH tunnel to connect to the simulator.

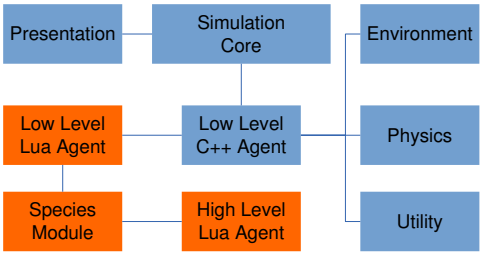


Fig. 1. System overview diagram. Blue is compile-time, orange are run-time modules

A. RANA's Component Modules

RANA comprises a simulation core, physics engine, environment model and agent models.

The **Simulation Core**, in C++, handles event storage and distribution and manages the timing and event processing of the simulation run. Events comprise external events generated by the agents and internal events used by the simulator core to model propagation delays. The core also defines the agent interface and exposes the physics and environment application programming interfaces (API). The physics engine performs calculations concerning sound propagation and attenuation over distance while the environment model comprises a terrain map (loaded as an image) that agents can query based on their location.

The **Modelling Interface** uses the APIs provided by the core to exchange events and access physics and environmental computations. However, models themselves are written in Lua, a small modern efficient functional scripting language. One can program directly against the exposed APIs using Lua. Alternatively, the Lua interface permits loading of Lua libraries and modules, making it possible to abstract model design — for instance one can define a species module, which individual agents load and further specialise; or construct a library to handle particular kinds of environmental feature in

cooperation with the terrain map. In this manner one can make it easy for modellers to use the tool, by providing libraries of encapsulated functionality for non-programmer users to exploit.

The C++ to Lua interface in the simulator core is extensible, allowing modules to register functionality to expose to the Lua agent models so that performance-critical components of models may easily be added to the C++ core. Data serialisation utilities allow the core and agents to exchange Lua data structures if desired. Lua execution speed is optimised by means of the impressive LuaJIT [13] just-in-time compiler without compromising Lua's clarity, run-time malleability or functional nature.

The Lua agents are loaded by the simulator at run-time and can be modified during a run (for instance to spawn new individualised agents).

B. Levels of Timing Precision

The core operates with two levels of precision: the micro-level, on which events are timed, processed and delivered to agents; and the macro-level on which agent dynamics typically operates. Agents respond to external events they receive using micro-level timing precision. However, the simulator core also queries agents once per macro-step so they can generate spontaneous action.

C. Performance

The C11 standard is used to create a high performance, cross-platform simulation core that takes advantage of modern threading interfaces to improve execution speed. Using Lua as modelling language gives its advantages of speed, efficiency and power as well as just-in-time compilation. The code has been optimised to devote as much CPU time as possible to event handling.

IV. Examples of RANA at Work

To illustrate the flexibility and utility of RANA, three different simulations are presented here. The first, mining robots, was developed to be part of a Ph.D. and M.Sc. level MAS course at the University of Southern Denmark. The second simulation tests the precision of RANA's event handling. The third is a state-based synchronising firefly model whose behaviour is illustrated using RANA's visualisation tools.

A. The Mining Robot Simulation

A group of agents is tasked with locating and collecting "ore" in a simulated world. There are three kinds of agent each with its own function:

- Explorers which roam the map looking for ore, the coordinates of which they transmit to nearby agents.
- Miners, which receive ore positions from explorers. They have limited memory and can only remember a fixed number of positions.
- Bases, which serve as recharge stations and ore caches for the explores and miners.

The world is a torus comprising 200×200 squares in which ore is distributed at random (7% of squares). The simulation is turn-based: each agent can perform one action per macro-step, where an action is moving one square, performing a scan etc.

The task was assigned to the summer course students, who used various MAS techniques to solve the problem. Two very different solutions were a design relying on a distributed blackboard system and a design using local repulsion and information-relaying to maximise efficiency. Figure 2 shows the world map displayed by RANA's visualisation tools after a simulation run of the second model.

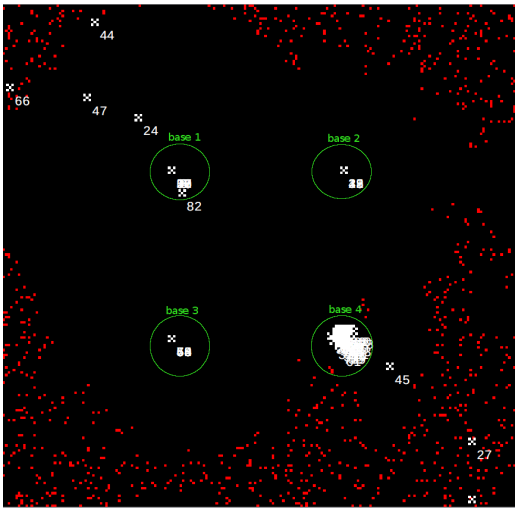


Fig. 2. Simulation setup for the mining robots: red pixels are ore, white x's are agents.

B. A Timing Test Agent Simulation

RANA advertises high-precision event timing. To test that claim, four agents set up in a square with an 80 m side pass a signal in sequence around the square. Events travel at 343 m/s. Agent 1 timestamps when it receives an event, i.e. each time the signal has traversed all four sides of the square. The recorded timestamps are stored and compared against the theoretical time in tableII.

TABLE II
Precision loss table. The delta values are the measured minus the theoretical timestamps

N	Time Delta[s]	Percentage Delta
1	6.4×10^{-6}	0.000685
5	3.6×10^{-5}	0.000771
500	0.0037	0.000792
10000	0.1479	0.000792
100000	0.7394	0.000792

The precision loss is very small: e.g., with a 400000 agent event relay, across a distance of 3200 km, the simulation delay error is 0.74 s (in 2.6 hours). The main reason for

the precision loss is quantisation error: each receive and response action takes 1 simulation micro-step. Ten runs yield identical results: the simulation core is deterministic. One can increase simulation precision, e.g. for simulation of faster event propagation such as radio communication, by adjusting the macro- and micro-level parameters.

C. The Synchronising Firefly

Using work on *Pteroptyx malaccae* fireflies [14] combined with a proposed model of insect communication [4], the third example is a simple state-based firefly model (illustrated in figure 3) which exhibits behaviour similar to the real insect. Using both the live view and the event visualiser it is possible to observe the emergent 'singing' of the agents. The fireflies will try to match the phase and frequency of other agents within their visual range.

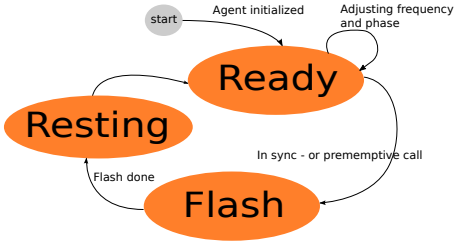


Fig. 3. The state-based model of the firefly. Every time the flash state is entered a 'flash' event is emitted.

This simple model achieves some resemblance to the real system. However, it is fragile and the emergent synchronisation can easily be disturbed by inserting agents exhibiting random frequency flashing. Without disturbing elements present, the model can achieve 70% of flashes in synch with another 500 agents (out of a 1000); inserting ten disrupter agents reduces the success rate to less than 40%.

V. Conclusion

RANA, a new MAS simulation tool optimised for precisely-timed models driven by broadcast events has been developed. The tool is portable, versatile and can handle large-scale and high-precision models. Its core is C++ while models programmed in Lua provide an approachable but very efficient high-level interface for modellers. Post-processing is possible both for analysis and visualisation purposes. Application of RANA to three different scenarios, briefly described, illustrates the tool's flexibility.

VI. Further Work

Our development of RANA has focussed on animal communication modelling. Future developments include: modelling animal flocks with a peer-to-peer social structure, which implies optimisation for large simulations with low macro-precision; porting the tool to HPC environments with decentralised event processing while retaining precision and accuracy; taking advantage of the Lua models to run individuals on small devices such as mobile phones which can be substituted for actual animals to test models' veracity; support for checkpointing is available, but needs further development.

Acknowledgements

The authors would like to thank Daniel Bjerring Jørgensen, Adam Czerwinsky and Kristjan Lundin for their data and their agent models of mining robots. It would not have been possible to establish the requirements needed to develop RANA without the help of biologist and sound communication expert Jakob Christensen-Dalsgaard. The work presented was funded by SDU's Centre for BioRobotics.

References

[1] D. Jones, R. Jones, and R. Ratnam, "Calling dynamics and call synchronization in a local group of unison bout callers," *Journal of Comparative Physiology A*, vol. 200, no. 1, pp. 93–107, 2014.

[2] M. Wooldridge and N. Jennings, "Agent theories, architectures, and languages: A survey," in *Intelligent Agents* (M. Wooldridge and N. Jennings, eds.), vol. 890 of *Lecture Notes in Computer Science*, pp. 1–39, Springer Berlin Heidelberg, 1995.

[3] P. Davidsson, "Agent based social simulation: A computer science view," *Journal of artificial societies and social simulation*, vol. 5, no. 1, 2002.

[4] M. D. Greenfield, M. K. Tourtellot, and W. A. Snedden, "Precedence effects and the evolution of chorusing," *Proceedings of the Royal Society of London. Series B: Biological Sciences*, vol. 264, no. 1386, pp. 1355–1361, 1997.

[5] I. Aihara, T. Mizumoto, T. Otsuka, H. Awano, K. Nagira, H. G. Okuno, and K. Aihara, "Spatio-temporal dynamics in collective frog choruses examined by mathematical modeling and field observations," *Scientific reports*, vol. 4, 2014.

[6] N. R. Jennings, K. Sycara, and M. Wooldridge, "A roadmap of agent research and development," *Autonomous Agents and Multi-Agent Systems*, vol. 1, pp. 7–38, Jan. 1998.

[7] S. Tisue and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in *in International Conference on Complex Systems*, pp. 16–21, 2004.

[8] R.-C. Damaceanu, "An agent-based computational study of wealth distribution in function of resource growth interval using netlogo," *Applied Mathematics and Computation*, vol. 201, no. 1, pp. 371–377, 2008.

[9] G. Lesins and K. Higuclu, "Agent based modelling of caribou environmental interactions in the Canadian arctic," *The International Environmental Modelling and Software Society*, 2010.

[10] E. Lee and P. I. Boulton, "The principles and performance of hubnet: A 50 mbit/s glass fiber local area network," *Selected Areas in Communications, IEEE Journal on*, vol. 1, no. 5, pp. 711–720, 1983.

[11] M. J. North, C. T. Nicholson, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with Repast Simphony," 2013. [Online; accessed 3-Feb-2014].

[12] S. Luke, C. Cioffi-Revilla, L. Panait, and K. Sullivan, "Mason: A new multi-agent simulation toolkit," in *Proceedings of the 2004 SwarmFest Workshop*, vol. 8, 2004.

[13] M. Pall, "The luajit project," 2008.

[14] J. Buck and E. Buck, "Mechanism of rhythmic synchronous flashing of fireflies fireflies of southeast asia may use anticipatory time-measuring in synchronizing their flashing," *Science*, vol. 159, no. 3821, pp. 1319–1327, 1968.

Printed by: Print & Sign, SDU.